



STIC

4

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

VLSI PUBLICATIONS

VLSI Memo No. 89-521  
April 1989

## A Reconfigurable Arithmetic Processor

James Alexander Stuart Fiske

### Abstract

Achieving high rates of floating-point computation is one of the primary goals of many computer designs. Many high speed floating-point datapaths have been designed in order to address this problem. However, conventional designs often neglect the real problem in achieving high performance floating-point: providing the necessary I/O bandwidth to keep the high speed datapaths busy.

The Reconfigurable Arithmetic Processor (RAP) is an arithmetic processing node for a message-passing, MIMD concurrent computer. Its datapath is designed to sustain high rates of floating-point operations, while requiring only a fraction of the I/O bandwidth required by a conventional floating-point datapath. The RAP incorporates on one chip eight 4-bit serial, 64 bit floating-point arithmetic units connected by a switching network. By sequencing the switch through different patterns, the RAP chip calculates complete arithmetic formulas. By chaining together its arithmetic units the RAP eliminates the I/O bandwidth associated with storing and retrieving intermediate results, and reduces the amount of off chip data transfer.

This thesis describes and evaluates the RAP architecture. It presents two important aspects of the chip design: the control logic design, and the schematic level design of the RAP datapath. The RAP datapath design includes the design of two 4-bit serial floating-point units: an adder/subtractor unit and a multiplier unit. In order to use the RAP datapath, a compiler is developed that takes as input a list of mathematical expressions, and outputs a series of switch configurations to be used by the RAP to do the calculation.

On 23 benchmark problems, the RAP reduced both the on chip and off chip bandwidth requirements by an average of 64%, when compared the bandwidth required by a conventional arithmetic chip that does not exploit locality. Average floating-point performance is 3.40 Millions of Floating-point operations per second (MFlops).

DISTRIBUTION STATEMENT A

Approved for public release  
Distribution Unlimited

AD-A208 323

DTIC  
ELECTE  
MAY 24 1989  
S D

002

### Acknowledgements

Submitted to the Department of Electrical Engineering and Computer Science at MIT on December 16, 1988 in partial fulfillment of the requirements for the Degree of Master of Science in Electrical Engineering and Computer Science. This work was supported in part by a scholarship from Fonds pour la Formation de Chercheurs et l'Aide a la Recherche (Fonds FCAR). Also in part by the Defense Advanced Research Projects Agency under contracts N00014-88-K-0738, N00014-87-K-0825, and N00014-85-K-0124 and in part by a National Science Foundation Presidential Young Investigator Award with matching funds from General Electric Corporation and IBM Corporation

### Author Information

Fiske: Electrical Engineering and Computer Science, Room NE43-416, MIT, Cambridge, MA 02139. (617) 253-8473.

Copyright© 1989 MIT. Memos in this series are for use inside MIT and are not considered to be published merely by virtue of appearing in this series. This copy is for private circulation only and may not be further copied or distributed, except for government purposes, if the paper acknowledges U. S. Government sponsorship. References to this work should be either to the published version, if any, or in the form "private communication." For information about the ideas expressed herein, contact the author directly. For information about this series, contact Microsystems Research Center, Room 39-321, MIT, Cambridge, MA 02139; (617) 253-8138.

# A RECONFIGURABLE ARITHMETIC PROCESSOR

by

James Alexander Stuart Fiske

Submitted to the  
Department of Electrical Engineering and Computer Science  
on December 16 in partial fulfillment of  
the requirements for the Degree of Master of Science in  
Electrical Engineering and Computer Science

## Abstract

Achieving high rates of floating-point computation is one of the primary goals of many computer designs. Many high speed floating-point datapaths have been designed in order to address this problem. However, conventional designs often neglect the real problem in achieving high performance floating-point: providing the necessary I/O bandwidth to keep the high speed datapaths busy.

The Reconfigurable Arithmetic Processor (RAP) is an arithmetic processing node for a message-passing, MIMD concurrent computer. Its datapath is designed to sustain high rates of floating-point operations, while requiring only a fraction of the I/O bandwidth required by a conventional floating-point datapath. The RAP incorporates on one chip eight 4-bit serial, 64 bit floating-point arithmetic units connected by a switching network. By sequencing the switch through different patterns, the RAP chip calculates complete arithmetic formulas. By chaining together its arithmetic units the RAP eliminates the I/O bandwidth associated with storing and retrieving intermediate results, and reduces the amount of off chip data transfer.

This Thesis describes and evaluates the RAP architecture. It presents two important aspects of the chip design: the control logic design, and the schematic level design of the RAP datapath. The RAP datapath design includes the design of two 4-bit serial floating-point units: an adder/subtractor unit and a multiplier unit. In order to use the RAP datapath, a compiler is developed that takes as input a list of mathematical expressions, and outputs a series of switch configurations to be used by the RAP to do the calculation.

On 23 benchmark problems, the RAP reduced both the on chip and off chip bandwidth requirements by an average of 64%, when compared the bandwidth required by a conventional arithmetic chip that does not exploit locality. Average floating-point performance is 3.40 Millions of Floating-point operations per second (MFlops).

Thesis Supervisor: William J. Dally

Title: Assistant Professor of Electrical Engineering and Computer Science

**Keywords:** Floating-Point, Bandwidth, Serial Arithmetic, Locality, J-Machine.

## Acknowledgments

Many people contributed in many ways to the completion of this Thesis, and I would like to thank them all whole heartedly.

First of all, thanks to my Thesis advisor Bill Dally for providing an immense amount of knowledge, technical expertise, and guidance which was invaluable to this work. Thanks for providing the sustained enthusiasm which helped carry me through till the end. Thanks also for helping me increase my well-traveledness by sending me to Banff and Hawaii.

The other members the CVA group were also a great help to me. Thanks to Andrew Chien for being the eternal skeptic, and always making me think twice about what I say. Thanks to Scott Wills for always taking time out to help me with NS and for being such a good humored guy. Thanks to Peter Nuth for founding the much needed "Hurting Dudes Club" and for his faithful late night presence and conversation at the lab. Thanks to John Keen for getting me to take sculling, and for getting me to go to surprise birthday parties. A special thanks to the original Yogurtheads of the group: thanks to Paul Song for his friendship, and his amusing optimism about how long things will take him to do, and to Brian Totty for his friendship, great sense of humor, and his deep appreciation of my singing. Thanks to all the other members of the group, new and old, including Soha Hassoun, Rich Lethin, Jerry Larivee, and Waldemar Horwatt. I am also indebted to Petr Spacek and Josef Shaoul for all the work that they did in designing the fixed point RAP (of course they left me to test it which wasn't very nice of them, but I'll overlook that).

Thanks to my friends, and ultimate Yogurtheads, Carlos Noack and Adam Ciaramicoli. Thanks for the Boston adventures we've had together, the good laughs, the good talks, and the Yogurthead songs composed and sung.

A special thanks to my entire family, especially my parents. Thanks for providing all the encouragement, love, and support that was necessary for me to pursue my formal education throughout these many years. Perhaps more importantly, thanks for participating so actively in my more informal education over the years.

Finally, and mostly, thank God!

|                       |                                     |
|-----------------------|-------------------------------------|
| Accession For         |                                     |
| NTIS - CRA&I          | <input checked="" type="checkbox"/> |
| DTIC TAB              | <input type="checkbox"/>            |
| Unannounced           | <input type="checkbox"/>            |
| Justification         |                                     |
| By <i>lt. on file</i> |                                     |
| Distribution          |                                     |
| Availability Codes    |                                     |
| Dist                  | Avail and/or Special                |
| A-1                   |                                     |



*To Mom and Dad*

# Contents

|          |                                                     |           |
|----------|-----------------------------------------------------|-----------|
| <b>1</b> | <b>Introduction</b>                                 | <b>7</b>  |
| 1.1      | The I/O Bandwidth Problem . . . . .                 | 7         |
| 1.2      | The RAP . . . . .                                   | 8         |
| 1.2.1    | Performance . . . . .                               | 10        |
| 1.3      | Background . . . . .                                | 11        |
| 1.3.1    | The J-Machine . . . . .                             | 11        |
| 1.3.2    | Arithmetic . . . . .                                | 11        |
| 1.3.3    | Another Approach . . . . .                          | 13        |
| 1.3.4    | The Scheduling Problem . . . . .                    | 14        |
| 1.4      | Thesis Overview . . . . .                           | 14        |
| <b>2</b> | <b>Architecture</b>                                 | <b>16</b> |
| 2.1      | An example . . . . .                                | 17        |
| 2.2      | Using the RAP . . . . .                             | 19        |
| 2.2.1    | A System Perspective . . . . .                      | 19        |
| 2.2.2    | Messages . . . . .                                  | 21        |
| 2.3      | Block Diagram . . . . .                             | 24        |
| 2.3.1    | Arithmetic Units . . . . .                          | 26        |
| 2.3.2    | Switch . . . . .                                    | 27        |
| 2.3.3    | Using the Datapath . . . . .                        | 29        |
| 2.4      | Comparison to Another Approach . . . . .            | 34        |
| 2.4.1    | Parallel Arithmetic vs. Serial Arithmetic . . . . . | 34        |
| 2.4.2    | Register File vs. Switch . . . . .                  | 39        |
| 2.5      | Summary . . . . .                                   | 41        |
| <b>3</b> | <b>RAP Simulation and Control Logic</b>             | <b>42</b> |
| 3.1      | Simulator Description . . . . .                     | 43        |
| 3.1.1    | Simulator Organization . . . . .                    | 43        |
| 3.2      | Control . . . . .                                   | 44        |
| 3.2.1    | Input Control . . . . .                             | 47        |
| 3.2.2    | Output Control . . . . .                            | 52        |
| 3.2.3    | Switch Control . . . . .                            | 52        |

|          |                                                               |            |
|----------|---------------------------------------------------------------|------------|
| 3.2.4    | Network Control . . . . .                                     | 55         |
| 3.3      | Summary . . . . .                                             | 57         |
| <b>4</b> | <b>Hardware Design</b>                                        | <b>58</b>  |
| 4.1      | Numbering System . . . . .                                    | 59         |
| 4.1.1    | Format . . . . .                                              | 59         |
| 4.1.2    | Normalized Numbers . . . . .                                  | 60         |
| 4.1.3    | Overflow and Underflow . . . . .                              | 61         |
| 4.1.4    | Zero Representation . . . . .                                 | 61         |
| 4.2      | Design of a 4-bit Adder . . . . .                             | 62         |
| 4.3      | Floating-Point Units . . . . .                                | 66         |
| 4.3.1    | Floating-Point Adder/Subtractor . . . . .                     | 67         |
| 4.3.2    | Floating-Point Multiplier . . . . .                           | 71         |
| 4.3.3    | Area Estimates . . . . .                                      | 76         |
| 4.4      | Other Hardware Components . . . . .                           | 77         |
| 4.5      | Hardware Improvements . . . . .                               | 77         |
| 4.6      | RAP Fixed-Point Prototype . . . . .                           | 80         |
| 4.7      | Summary . . . . .                                             | 85         |
| <b>5</b> | <b>Expression Compiler</b>                                    | <b>86</b>  |
| 5.1      | The Problem . . . . .                                         | 87         |
| 5.2      | The Algorithm . . . . .                                       | 88         |
| 5.2.1    | An Example . . . . .                                          | 89         |
| 5.2.2    | Sequencing Schedules . . . . .                                | 93         |
| 5.2.3    | Limiting the Search . . . . .                                 | 95         |
| 5.3      | Other Enhancements and other Approaches . . . . .             | 97         |
| 5.4      | Summary . . . . .                                             | 102        |
| <b>6</b> | <b>Performance</b>                                            | <b>104</b> |
| 6.1      | Evaluation Method . . . . .                                   | 105        |
| 6.1.1    | Assumptions . . . . .                                         | 105        |
| 6.1.2    | Benchmarks . . . . .                                          | 106        |
| 6.2      | Bandwidth Performance . . . . .                               | 108        |
| 6.3      | Floating-Point Performance . . . . .                          | 110        |
| 6.4      | Limits on Performance . . . . .                               | 113        |
| 6.4.1    | Limits on Bandwidth Performance . . . . .                     | 113        |
| 6.4.2    | Limits on Floating-Point Performance . . . . .                | 114        |
| 6.5      | Improving Performance . . . . .                               | 115        |
| 6.5.1    | Finding Better Methods . . . . .                              | 115        |
| 6.5.2    | Using Different Resource Configurations . . . . .             | 116        |
| 6.5.3    | Pipelining RAPs . . . . .                                     | 118        |
| 6.5.4    | Reorganizing Control and Operation of the Data Path . . . . . | 120        |
| 6.6      | Summary . . . . .                                             | 122        |

|                                            |            |
|--------------------------------------------|------------|
| <b>7 Conclusion</b>                        | <b>124</b> |
| 7.1 Summary . . . . .                      | 125        |
| 7.2 Future Work . . . . .                  | 127        |
| <b>A 4-Bit Adder Simulation and Layout</b> | <b>129</b> |
| A.1 Simulation . . . . .                   | 130        |
| A.2 4-bit Adder Schematics . . . . .       | 132        |
| A.3 4-bit Adder Layout . . . . .           | 158        |

# List of Figures

|      |                                                                             |    |
|------|-----------------------------------------------------------------------------|----|
| 1.1  | RAP Datapath . . . . .                                                      | 10 |
| 1.2  | J-Machine Configuration . . . . .                                           | 12 |
| 2.1  | 4-Point FFT Dataflow Graph . . . . .                                        | 18 |
| 2.2  | RAP Usage . . . . .                                                         | 20 |
| 2.3  | Message Formats . . . . .                                                   | 22 |
| 2.4  | RAP Block Diagram . . . . .                                                 | 25 |
| 2.5  | Switch Configuration . . . . .                                              | 28 |
| 2.6  | Computing a Single Problem Without Pipelining . . . . .                     | 30 |
| 2.7  | Computing a Single Problem Using Pipelining . . . . .                       | 31 |
| 2.8  | Computing Multiple Instances of the Same Problem Using Pipelining . . . . . | 32 |
| 2.9  | Parallel Arithmetic Using a Register File . . . . .                         | 35 |
| 2.10 | Parallel Arithmetic vs. Serial Arithmetic Efficiency . . . . .              | 37 |
| 2.11 | Register File Area vs. Switch Area . . . . .                                | 40 |
| 3.1  | Input Control Flow Chart (part 1) . . . . .                                 | 49 |
| 3.2  | Input Control Flow Chart (part 2) . . . . .                                 | 50 |
| 3.3  | Input Control Flow Chart (part 3) . . . . .                                 | 51 |
| 3.4  | Output Control Flow Chart . . . . .                                         | 53 |
| 3.5  | Switch Control Flow Chart . . . . .                                         | 54 |
| 3.6  | Network Input Control Flow Chart . . . . .                                  | 55 |
| 3.7  | Network Output Control Flow Chart . . . . .                                 | 56 |
| 4.1  | Floating-Point Format . . . . .                                             | 60 |
| 4.2  | General 4-bit Adder SPICE circuit . . . . .                                 | 63 |
| 4.3  | Manchester Carry Circuits . . . . .                                         | 65 |
| 4.4  | Stages of the Ripple Carry Circuitry . . . . .                              | 65 |
| 4.5  | Floating-Point Adder/Subtractor Block Diagram . . . . .                     | 68 |
| 4.6  | Steps in the Floating-Point Add . . . . .                                   | 69 |
| 4.7  | Floating-Point Multiplier Block Diagram . . . . .                           | 71 |
| 4.8  | Steps in the Floating-Point Multiply . . . . .                              | 72 |
| 4.9  | Mantissa Multiply Pipeline . . . . .                                        | 73 |
| 4.10 | Example Multiply of Two Positive Fractions . . . . .                        | 74 |

|      |                                                                                                        |     |
|------|--------------------------------------------------------------------------------------------------------|-----|
| 4.11 | Simplified Multiply Cell . . . . .                                                                     | 74  |
| 4.12 | Input and Output Register Cells . . . . .                                                              | 78  |
| 4.13 | Switch Cross Point . . . . .                                                                           | 79  |
| 4.14 | RAP Fixed-Point Prototype . . . . .                                                                    | 82  |
| 4.15 | Block Diagram of the Fixed-Point RAP . . . . .                                                         | 83  |
| 4.16 | Fixed Point RAP Datapath . . . . .                                                                     | 84  |
| 5.1  | DAG for the List of Expressions $(*(+ 3 X) (+ Y X)), (* (+ Z Y) (+ Z H)),$<br>$(*(+ H 2) 4)$ . . . . . | 90  |
| 5.2  | DAG Schedule (part 1) . . . . .                                                                        | 91  |
| 5.3  | DAG Schedule (part 2) . . . . .                                                                        | 92  |
| 5.4  | Tree of All Possible Schedules for a Level . . . . .                                                   | 94  |
| 5.5  | An Accumulation Tree . . . . .                                                                         | 99  |
| 5.6  | Accumulation Tree with Redundant Operations . . . . .                                                  | 100 |
| 5.7  | Eliminating Common Subexpressions . . . . .                                                            | 101 |
| 6.1  | An Alternative Switch . . . . .                                                                        | 119 |
| A.1  | SPICE Plots for Different 4-Bit Adders . . . . .                                                       | 131 |

## List of Tables

|     |                                                                                         |     |
|-----|-----------------------------------------------------------------------------------------|-----|
| 4.1 | Overflow/Underflow Conventions for the Add, Subtract, and Multiply Operations . . . . . | 62  |
| 4.2 | 4-bit Adder Delay and Area . . . . .                                                    | 64  |
| 4.3 | Basic Cell Area Estimates . . . . .                                                     | 76  |
| 4.4 | Floating-Point Unit Area Estimate . . . . .                                             | 77  |
| 6.1 | Benchmarks used to Evaluate Performance . . . . .                                       | 107 |
| 6.2 | RAP Bandwidth Performance . . . . .                                                     | 109 |
| 6.3 | RAP Floating-Point Performance . . . . .                                                | 111 |
| 6.4 | Comparison of Actual Performance to Ideal Performance . . . . .                         | 112 |
| 6.5 | Optimum # of Configurations per Method vs. Number of Input Operands .                   | 120 |

# Chapter 1

## Introduction

*Let us go singing as far as we go: the road will be less tedious.*

— VIRGIL, in *Eclogues*, IX, l.64

*What experience and history teach is this — that people  
and governments never have learned anything from history,  
or acted on principles deduced from it.*

— GEORGE WILHELM FRIEDRICH HEGEL in *Philosophy of History* (1832)

### 1.1 The I/O Bandwidth Problem

The problem in building fast arithmetic chips is not building fast arithmetic circuits but rather supplying the necessary I/O bandwidth. For example, a conventional 64 bit-parallel floating-point adder or multiplier pipe computing at 20MFlops (Millions of Floating-point operations per second) requires an I/O bandwidth of 3.8Gbit/sec. This rate of I/O is very difficult to achieve with anything less than dedicated lines and a continuous stream of data.

The problem is only getting worse: today it is possible to build a pipelined bit-parallel 100MFlop floating-point adder or multiplier, but be unable to exploit more than a small fraction of its power due to insufficient I/O bandwidth.

The I/O bandwidth problem occurs at two different levels: off chip I/O and on chip I/O. Off chip I/O is the most severe problem because of packaging limitations. Off chip capacitances are orders of magnitude larger than on chip capacitances (10 pf compared to 0.01 pf) which slows down the propagation of signals. Also, the number of pins on a chip is limited due to physical constraints. Although packaging technology is improving, the inherent physical limitations prevent off chip bandwidth from achieving levels that are possible on chip, and ways must be found to limit off chip I/O requirements.

On chip there can be a bandwidth problem between storage (memory and registers), and logic circuitry. The problem is caused by high capacitance bus and memory lines which limit the speed at which data can be moved between storage and logic. The problem is much less severe than the off chip case since there are many mechanisms for dealing with the problem: multiple busses, multi-ported registers and memory, and sophisticated sensing circuitry. These approaches to solving the problem can mean a substantial increase in chip area.

Since technology improvements are not eliminating the I/O problem, it is important to explore architectural solutions to both the off chip and the on chip I/O bandwidth bottlenecks. The architecture of the Reconfigurable Arithmetic Processor (RAP) addresses both these problems in the case of high speed floating-point arithmetic.

## 1.2 The RAP

The RAP is a CMOS, 64 bit, floating-point arithmetic chip. It is designed to sustain high rates of floating-point operations, while requiring only a fraction of the I/O bandwidth of a

conventional arithmetic chip. To do this the RAP allows the direct calculation of complete expressions that contain several adds, subtracts, and multiplies.

The RAP uses serial arithmetic. Bit-serial arithmetic implementations are more area efficient than bit-parallel implementations in that they require a much smaller amount of chip area. This area efficiency is due to the use of narrow datapaths rather than wide datapaths. The savings in area is not without cost: serial implementations are slower than parallel implementations because bits must be clocked sequentially into and out of the circuit. Results cannot "flow through" to the output as in parallel implementations.

The reduced area requirements of serial arithmetic allow several Arithmetic Units (AUs) to be put on a single chip. Having narrow serial datapaths also allows the implementation of an area efficient switching network that can be used to route data between AUs. Although a single serial unit is slower than a parallel implementation, the RAP makes up for this by exploiting the functional parallelism achieved by having several units on one chip: instead of using a single 32MFlop bit parallel unit, eight 4MFlop bit serial units running in parallel are used. Performance is then determined by the extent to which this parallelism can be exploited, which in turn is dependent on the structure of the problem.

The reconfigurable RAP datapath shown in Figure 1.1 consists of a number of 4-bit serial AUs, a switch, input registers, and output registers. Data is first shifted through the switch and gets routed to the appropriate AUs. Intermediate results are fed back into the switch which is reconfigured to allow the next stage of the computation to take place. When the computation is complete the results are sent to the output registers. A compiler has been written that compiles mathematical expressions into the successive switch configurations needed to perform the calculation.

At a higher level, the RAP has a message passing interface that allows it to be integrated into the J-Machine[8], a message passing concurrent computer system. A RAP is sent messages that define equations as a sequence of switch configurations, which are stored in

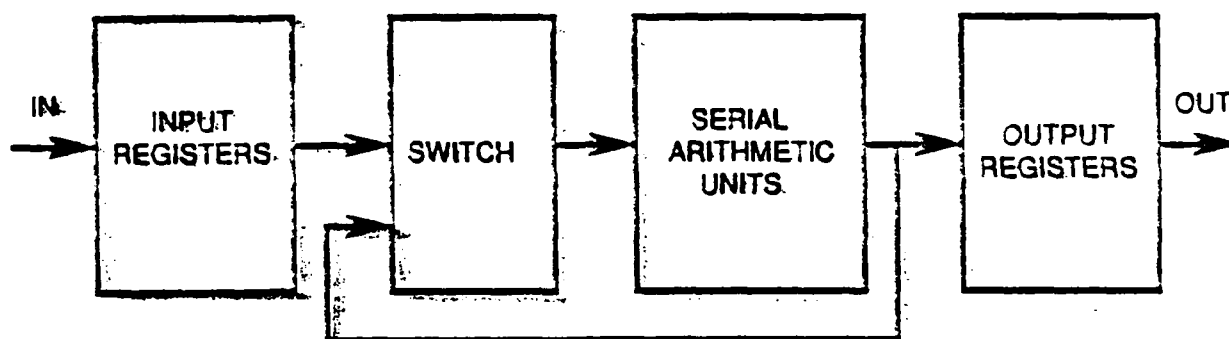


Figure 1.1: RAP Datapath.

local memory. Subsequent messages use these stored configurations to evaluate the equation. Mechanisms are included that allow the pipelining of several RAPs so that the output of one RAP can be used as the input to another.

### 1.2.1 Performance

Two serial floating-point AUs, one an adder/subtractor, the other a multiplier, were designed and have an expected performance of 1.57MFlops. In the current design, there is a large time gap between when two successive problems can enter the AUs. Straightforward modifications to the AU design would eliminate these wasted clock cycles and allow better pipelining of problems. This would increase the AU performance to 4.70MFlops. When used in the RAP, some extra time is needed to change switch configurations, so that the AUs would have a peak performance of 4MFlops. This gives a peak performance of 32MFlops for a RAP containing four add/subtract units and four multiply units.

The average <sup>1</sup> floating-point performance achieved for 23 benchmark problems that were simulated is 3.40MFlops or 11% of the peak performance. More importantly, the off chip I/O bandwidth required to sustain the computation rate is reduced on average by 64%.

<sup>1</sup>Throughout this thesis, "average" refers to the harmonic mean whenever the quantity in question is a "rate", such as a floating-point rate or a bandwidth rate.

This is when compared to the bandwidth required by a conventional chip where no locality is exploited. When compared to a conventional chip augmented by a register file used to keep intermediate results from going off chip, the on chip I/O bandwidth required between storage and logic in the RAP is also reduced by 64%. The RAP approach also results in a more area efficient implementation.

## 1.3 Background

### 1.3.1 The J-Machine

The RAP chip is being designed as a part of the J-Machine [8], a message passing concurrent computer system under development at MIT. This system is based on a mesh routing network that connects a collection of processing nodes, and uses wormhole routing techniques to reduce message latency to approximately  $2\mu\text{s}$  for a 200 bit message on a 4K node network [9]. Each single-chip node includes both the network communication hardware and a processor. The RAP chip is one node type that can fit into the network "slots", as shown in Figure 1.2. It includes the necessary control mechanisms and message handling capabilities to fit into the system. The RAP borrows several ideas that were first developed in the Message Driven Processor (MDP) [7], the general purpose computing node for the system. In particular the RAP executes messages directly, reducing message interpretation overhead, and it makes use of the same network communication scheme [6, 9].

### 1.3.2 Arithmetic

Many computer applications in such areas as analog circuit simulation, N-body problems, finite element analysis, digital signal processing, and three dimensional graphics require large amounts of floating-point computing power [32]. To satisfy this demand, many spe-

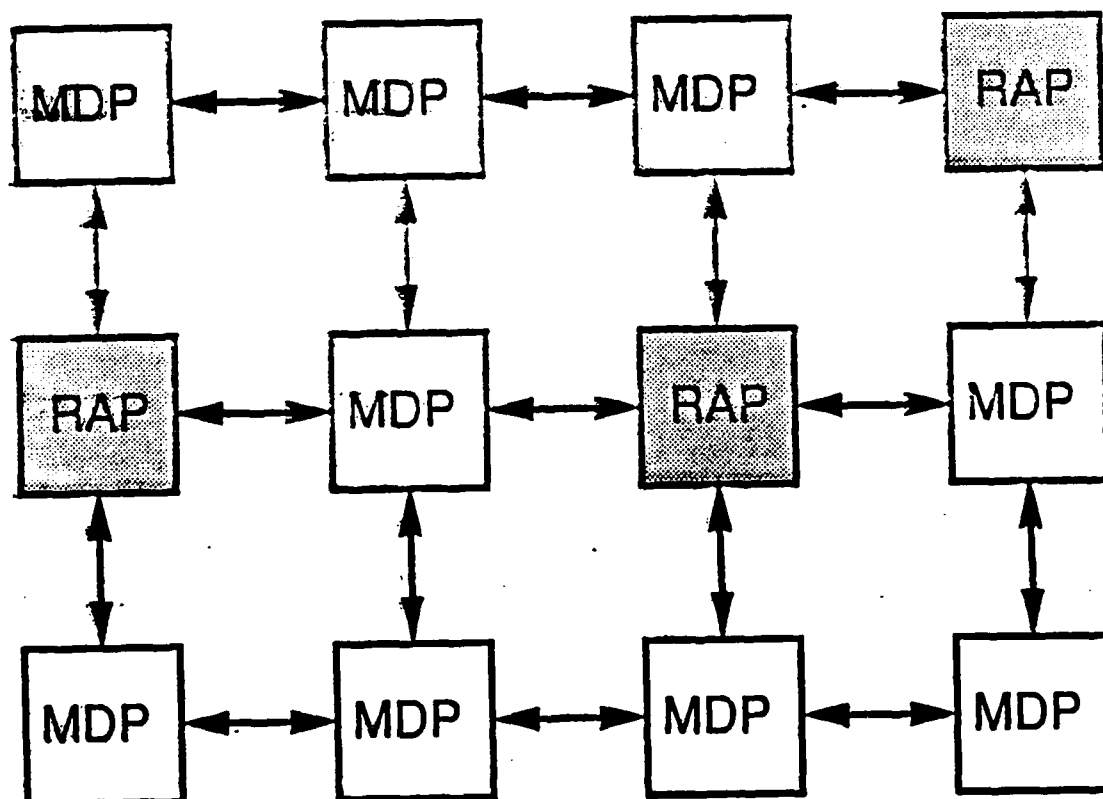


Figure 1.2: J-Machine Configuration

cial purpose board level and chip level arithmetic processors have been built [14, 26, 4]. Approaches range from math coprocessors that act as extensions to a main processor (e.g. the Intel 80387 and the Motorola MC68881) to dedicated math processors designed for specific applications. In most cases these processors are implemented using a bit-parallel approach. Because of this approach, implementations are expensive in terms of silicon area and only one or two floating-point units can be put on a single chip.

The area efficiency of serial arithmetic allows several floating-point units to be put on a single chip. Serial arithmetic has been used in many Digital Signal Processing applications [30, 24]. The idea of exploiting functional parallelism using serial fixed-point arithmetic has been used in this area [25]. Many algorithm alternatives exist for serial arithmetic implementation [5, 23, 16, 29, 35].

### 1.3.3 Another Approach

Another approach to the I/O bandwidth problem is to use an on chip register file to store operands and intermediate results [4]. The register file serves the same function as a switch, selecting data to be input to each function unit during each pipeline time slot. The register file performs this switching both by storing data to move it to a different time slot, and by multiplexing many registers into each register file port. The serial switch in the RAP eliminates the need for storage and simplifies the multiplexing. The resulting switch is smaller, both because it is serial and because it contains no storage. The switch is also simpler to control: switch configurations are changed each word time (a word time corresponds to the time it takes to clock a 64 bit operand serially through the switch), while register file addresses must be changed each clock cycle. The slow control signals allows the switch to operate faster than a comparable register file. A more detailed comparison of the register file approach and the approach used in the RAP is found in Chapter 2.

### 1.3.4 The Scheduling Problem

In the RAP the operations in an expression must be scheduled on the functional units. This scheduling is done subject to a number of constraints, including the number of functional units available and the connectivity of the switch. The scheduling of partially ordered tasks on limited and shared resources has been studied in different contexts including operations research, microprogramming, and parallel computing. In particular, compilers for VLIW processors [10, 21] that schedule operations on multiple functional units, deal with a problem very similar to the scheduling problem on the RAP. The approach used in these compilers is one called list scheduling which involves keeping a list of schedulable operations, and using heuristics to help in determining which operations should be scheduled first.

The technique used to schedule expressions on the RAP involves a depth first search that uses various tree search pruning techniques such as branch-and-bound [33, 37]. Heuristics similar to those used in list scheduling are used to determine which operations should be scheduled first.

## 1.4 Thesis Overview

This Thesis describes the RAP architecture, presents the design of the logic required to control the RAP as well as the circuit design of the RAP datapath, describes an expression compiler which maps expressions onto the RAP switch, and evaluates the RAP performance.

Chapter 2 describes the RAP architecture. How the RAP reduces the off chip bandwidth requirements to do floating-point computations is illustrated with an example. How the RAP works and how it is used and controlled with messages is discussed. The RAP's block diagram is described in detail, and the architecture is compared to the use of a multi-ported register file to exploit locality and reduce bandwidth requirements.

Chapters 3 and 4 deal with two of the most important aspects of the chip design: the design of the control logic and the design of the datapath hardware. In Chapter 3 a RAP simulator is used to design and verify the RAP control logic. This simulator is also used in Chapter 6 to help evaluate performance. Chapter 4 presents the hardware design of the RAP datapath, in particular the design of the floating-point functional units and the design of the switch, which are the most critical components in determining the performance of the RAP.

In order to be able to use the RAP arithmetic expressions or sets of expressions must be mapped into a series of RAP switch configurations. A compiler which performs this mapping is described in chapter 5.

Chapter 6 presents a performance evaluation of the RAP. The expression compiler of Chapter 5 is used to map a number of benchmarks onto the RAP and performance is evaluated in terms of the bandwidth required, and in terms of floating-point rates achieved. The factors that limit performance are discussed, and various schemes that can be used to improve performance are examined.

Finally, the results of the thesis are summarized and a few open research issues are discussed in chapter 7.

## Chapter 2

# Architecture

*Experience has shown that to be true which Appius says in his verses, that every man is the architect of his own fortune.*

— SALLUST, in *Speech to Caesar on the State*, sec. 1

*Do all the good you can,  
By all the means you can,  
In all the ways you can,  
In all the places you can,  
At all the times you can,  
To all the people you can,  
As long as ever you can.*

— JOHN WESLEY, *John Wesley's Rule*

In this chapter a complete description of the RAP is given. In section 2.1 an example problem is shown which illustrates the RAP operation and how it reduces bandwidth requirements. Section 2.2 describes the different ways of using the RAP within a J-Machine system and describes the messages used to control the RAP. Section 2.3 gives a detailed block diagram of the RAP. Finally, section 2.4 compares the approach used in the RAP to

reduce off chip bandwidth to the approach used in more conventional design, which involves using an on chip register file.

## 2.1 An example

In order to illustrate how the RAP uses functional parallelism to exploit the locality and the concurrency found in mathematical equations, the calculation a 4-point Fast Fourier Transform (FFT) [28, 31] is examined. The 4-point FFT dataflow graph is shown in Figure 2.1. It consists of 12 multiplies and 22 additions used to calculate the real and imaginary parts of the 4 output results. This graph is evaluated by a RAP as follows: First a "method" describing the schedule for each level of the calculation is stored in the RAP memory. Then a message is received containing the 14 input variables necessary for the computation. Assuming an ideal setting, the RAP successively runs through each level of the calculation as described by the method, exploiting functional parallelism by doing all operations of a given level in parallel. Finally it sends a message containing the results to the appropriate destination.

In a realistic setting, determining the successive configurations of a method involves a scheduling problem, since the RAP may not have enough AUs to perform all possible concurrent operations at once. The RAP has four adders/subtractors and four multipliers. At any given time there may not be enough AUs to begin all operations that have their operands ready. Furthermore, the switch may limit the operations that can be scheduled if it prevents two operands from both reaching the same AU. The scheduling problem is discussed in detail in Chapter 5.

The off chip I/O bandwidth required is reduced to 25% of the bandwidth required by a conventional bit-parallel arithmetic chip. A conventional arithmetic chip would require  $34 \times 3 = 102$  word transfers, where 34 corresponds to the number of operations, and 3 corresponds to the two words of input data and one word of output data for each opera-

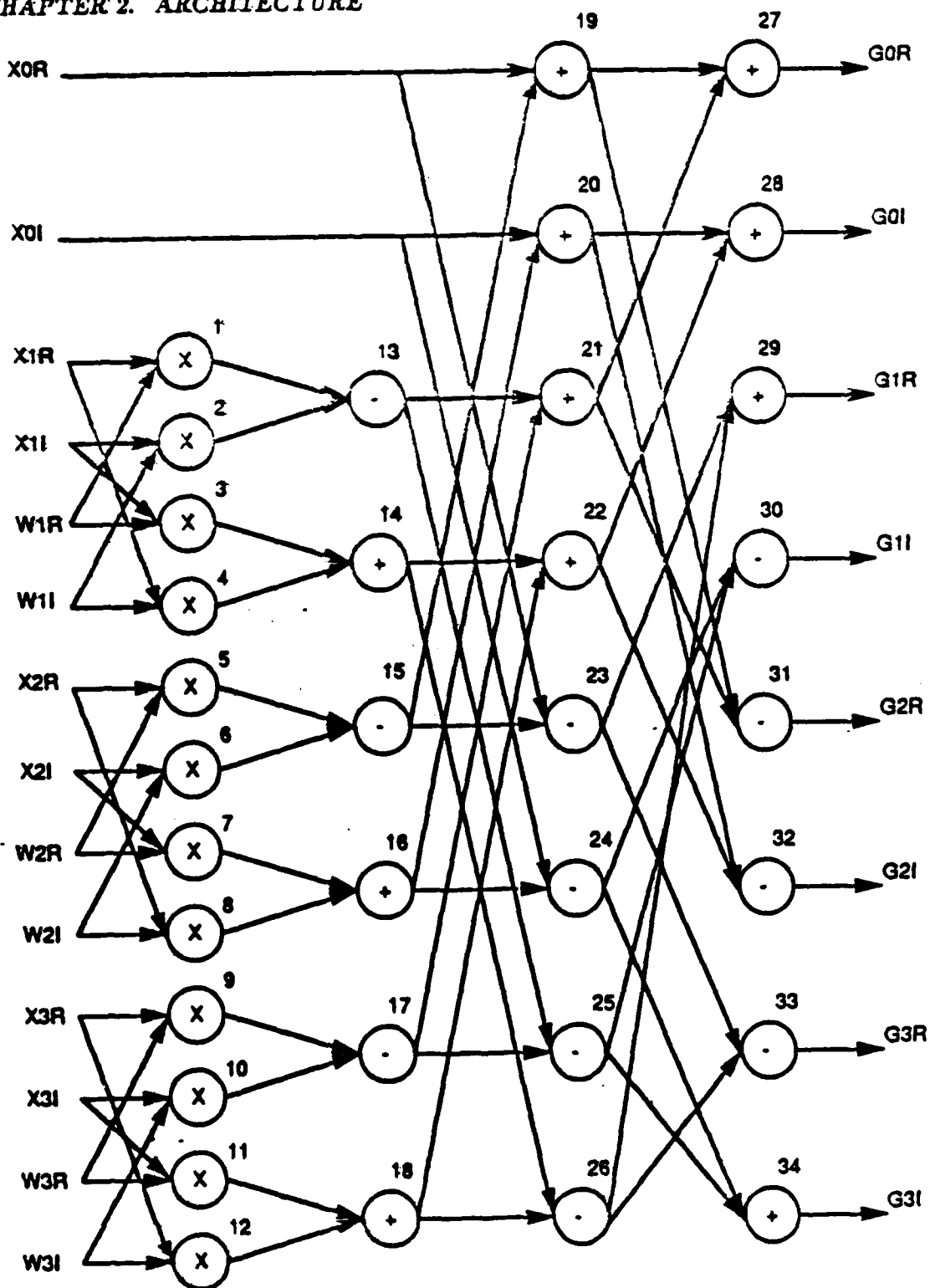


Figure 2.1: 4-Point FFT Dataflow Graph

tion. Using a RAP, only 26 words must be transferred on and off chip, consisting of 14 input operands, 8 output results, and 4 words of overhead information. This reduced I/O bandwidth makes it possible for a communications network to keep the chip busy.

Note that only the data I/O is considered here. It is assumed that the 4X4FFT method has already been stored in the RAP memory, so that there is no control overhead other than message overhead. The RAP is able to store several different methods in its memory.

## 2.2 Using the RAP

### 2.2.1 A System Perspective

Within the context of the J-Machine, a single RAP can be used to do a calculation, or several RAPs can be used at once to help speed up the calculation. When used by itself the RAP acts as a compute server which receives messages from MDPs, does the calculations requested by these incoming messages, and forwards the results to a specified MDP destination.

RAPs can also be used in combination to complete a calculation faster. One example of this is pipelining a computation through several RAPs, as shown in Figure 2.2b. Each RAP does one part of the computation, and feeds intermediate results to the next RAP which continues the computation. As in all pipelines, an effort must be made to match the pipeline stages: each stage should have roughly the same amount of work to do so that no stage holds up any other stage. The network must also be considered as a pipeline stage since it passes data between RAPs at a finite speed.

An extension of the idea of pipelining is the idea of forking shown in Figure 2.2c. In this case the work of a given pipeline stage is fanned out over several RAPs. The counterpart of a fork operation is the join operation, where the results from several RAPs are combined in

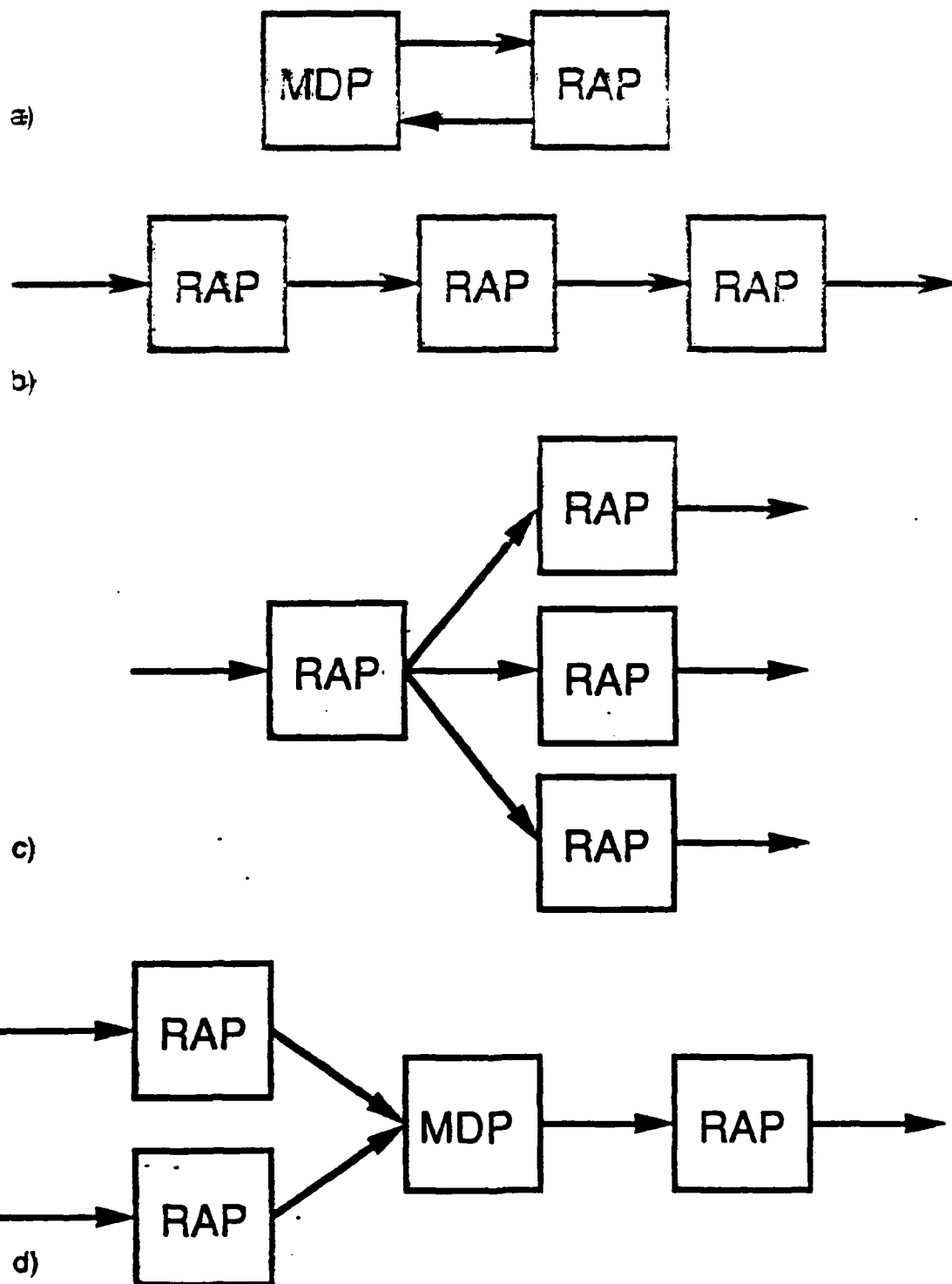


Figure 2.2: RAP Usage: a) RAP Used Alone, b) RAP Pipeline, c) RAP Fork Operation, d) RAP Join Operation

some way, before the computation is continued. This is shown in Figure 2.2d and requires an MDP to perform the synchronization and combining of results coming from different RAPs.

One simple mechanism based on forwarding templates is provided in the RAP which permits it to be used in all the ways described above. Templates and their use are described in the next section.

### 2.2.2 Messages

There are three types of messages that the RAP processes in order to support the types of operations described above:

1. **CONFIGURE AND EXECUTE (C+E).** This message causes operands to be loaded into the input registers, passed through one or more switch configurations, and then unloaded from the output registers. This is repeated for each set of operands in the message.
2. **STORE METHOD (SM).** This message is used to store a method in local memory so that it can be used by the C+E message. A method describes a sequence of switch configurations necessary to perform a calculation.
3. **STORE TEMPLATE (ST).** This message is used to store a template in local memory. A template contains forwarding information that allows the forwarding of results to a specified destination such as an MDP or another RAP.

Message formats are shown in Figure 2.3. The C+E message has METHOD-ID and TEMPLATE-ID fields that specify the method and template to be used. Method and template IDs are memory addresses that point at the first element of the method or template. Methods and templates must be sent to the RAP before the C+E commands that use them.

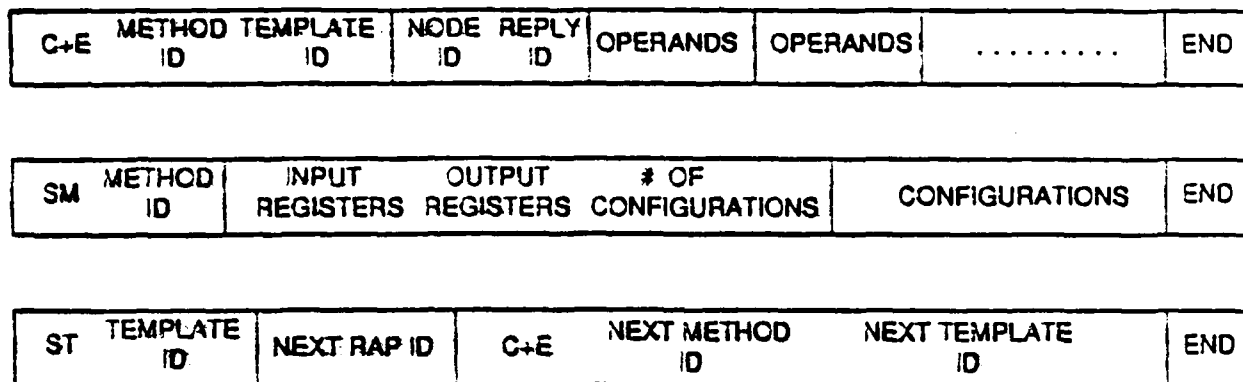


Figure 2.3: Message Formats

The C+E message also has NODE-ID and REPLY-ID fields that specify the ultimate destination of the results. The NODE-ID is the network address of a non-RAP node, and the REPLY-ID is a message header. These two fields are used in conjunction with template information to forward output results. The information contained in methods and templates is discussed in detail in the following sections.

### Methods

A method consists of all the information necessary to put operands through a sequence of switch configurations. It includes:

1. Which input registers to load with the operands.
2. Which output registers will contain the results.
3. The number of switch configurations that the operands are to go through.
4. A description of each of the configurations. Each configuration specifies the switch connectivity, and the functionality of the AUs (e.g. one bit might determine whether an AU does an add or a subtract).

The first three pieces of information are packed into one word (72 bits<sup>1</sup>) of the method description. Each configuration also takes one word. The number of operand sets that will be used with a given method is not included in the method description. It can be deduced from the end of message signal.

### Templates

A template is used to permit the cascading of several RAP chips. It contains information that allows the forwarding of the output data, in the form of a C+E message, to another RAP for further computation. A template consists of the address of the next RAP that the results are to be sent to, and the instruction (method and template) that is to be executed there.

Cascading of RAPs works as follows: the MDP sets up the pipeline by loading methods and templates into the appropriate RAP chips. Then a C+E message is sent to the first RAP in the pipeline, beginning the calculation. Each RAP uses its template to forward results to the next RAP in the pipeline. The return address, in the form of the NODE-ID and REPLY-ID, is passed from RAP to RAP until it is used at the last stage to get the results to their final destination (the use of the "default" template causes the results to be sent to the return address).

Templates are specified separately from methods. This allows different calculations to use common subroutines and permits a single calculation to distribute its work over several RAPs (i.e. do a fork operation). For example, a routine that multiplies all the elements of two vectors can be used by several different calculations. By using a different template to forward the result, such a routine can be used by itself or can be used in an inner product routine. A RAP can also use the different templates to divide up a problem, fanning data

---

<sup>1</sup>The MDP has a 36 bit word including a 32 bit data field and a 4 bit tag field. It is convenient to define the word size for the RAP to be 72 bits or two MDP words, since all operands are 64 bit floating-point numbers.

out to a number of different RAPs. In this case several RAPs contain the same method and the choice of templates distributes the work over these processors.

## 2.3 Block Diagram

Figure 2.4 shows a block diagram of the complete RAP consisting of the control blocks, the memories, and the datapath. There are four control blocks: input control, output control, switch control, and network interface control. Input control executes incoming messages, and controls the input to the datapath, and most memory operations. Output control is responsible for creating result messages in the output queue. Switch control is responsible for loading switch configurations at the correct time. Finally, network interface control is responsible for message reception and transmission. By dividing the control into these four different blocks the operations of receiving a message, loading operands, changing the switch configuration, unloading results, and sending a result message can be pipelined. Hardware interlocks resolve memory contention and provide feedback to prevent the queues from overflowing.

There are three memories on the chip: a main memory for holding templates and methods, an input queue, and an output queue. The input and output queues are 64 word memories with separate ports for the network and processor. The main memory (256 words) has separate input and output ports and is shared between the input control and the switch control, with priority given to the switch control.

The datapath consists of 16 input registers, a switch, a switch configuration register, a collection of 16 functional units, 16 output registers, and some buffer storage for the template. The 16 functional units consist of 4 add/subtract AUs, 4 multiply AUs, and 8 feedthroughs. The 8 feedthrough units are used to pass operands unchanged with a fixed delay.

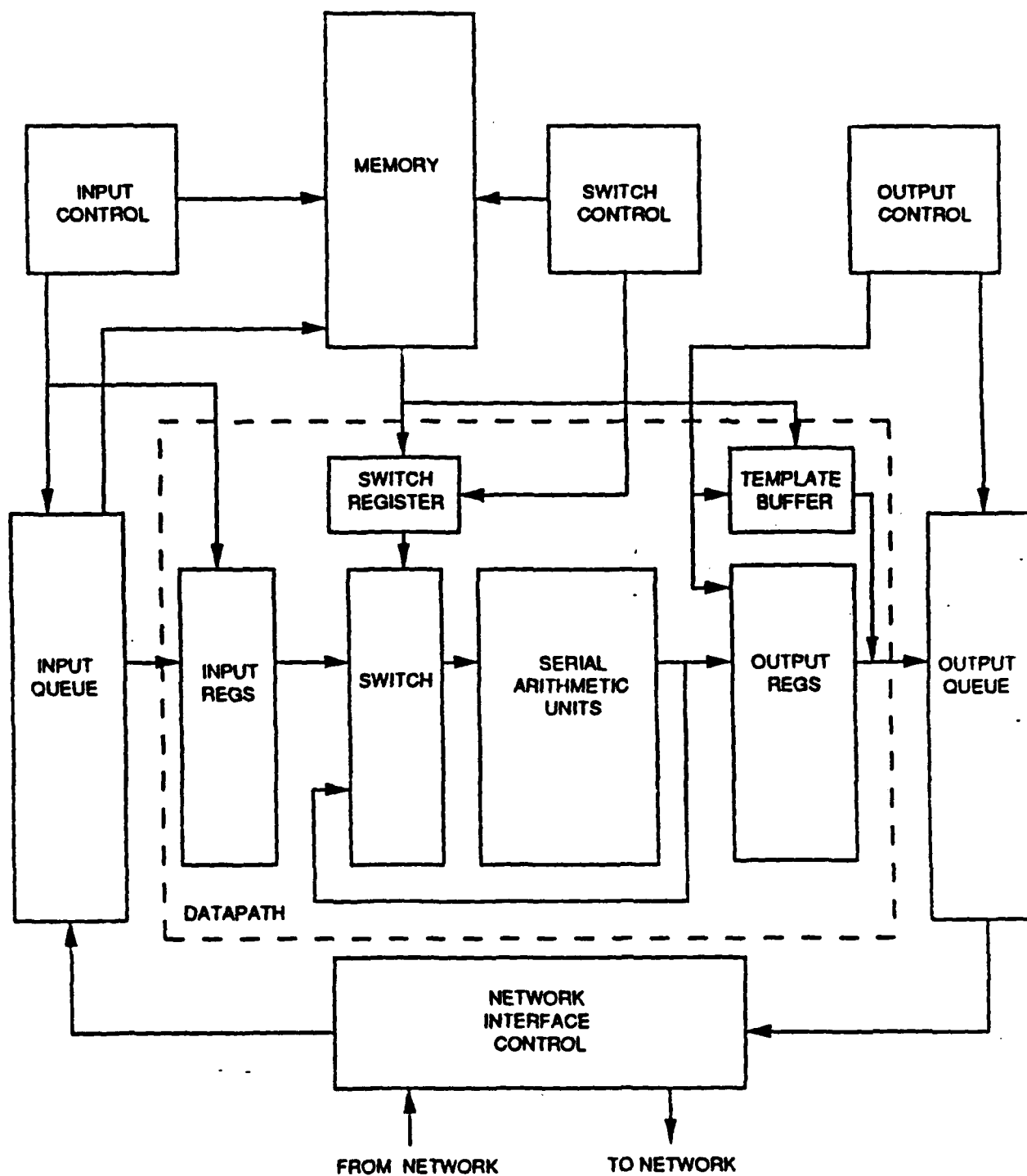


Figure 2.4: RAP Block Diagram

Initially, operands are loaded into the input registers from the input queue, and are shifted through the switch into the functional units. The outputs of the functional units feed back into the switch, allowing intermediate results to be routed back to the inputs of the functional units. The results are finally shifted into the output registers and then are unloaded into the output queue. The input and output registers perform parallel-serial and serial-parallel conversion respectively, and can be loaded or unloaded as the switch and functional units are busy computing another problem. In order to perform the routing of intermediate results, the switch is reconfigured at regular intervals by the switch control unit which reloads the switch configuration register. The message header information is taken from the appropriate template and is unloaded into the output queue before the output results.

### 2.3.1 Arithmetic Units

The RAP includes adders/subtractors, and multipliers. The design of these units is presented in Chapter 4. Their estimated performance is 1.57MFlops, with a floating-point adder/subtractor Requiring  $3.2M\lambda^2$  and a floating-point multiplier requiring  $5.6M\lambda^2$ .

The AUs are clocked at 80Mhz which is four times faster than the 20MHz clock used for the memory and control. It is convenient to define two types of cycles: a *minor cycle* corresponding to an AU clock cycle, and a *major cycle* corresponding to a memory cycle. It is also convenient to define a *word time* as the time required to shift a complete operand into an AU. Since the AUs are 4-bit serial, a word time corresponds to 16 minor cycles or 4 major cycles. The units have a latency of approximately three word times. During this time the exponent and mantissa are computed, and normalization is performed.

In the initial design there is also a three word latency between when two different problems can begin computing. However, with suitable design modifications <sup>2</sup>, this latency

---

<sup>2</sup>These modifications were not implemented due to time constraints. Chapter 4 discusses in more detail

could be reduced to one word time. This means that as many as three problem could be in the AUs at one time (one problem for each word of latency through the AU), increasing the performance of an individual AU to 4.70MFlops.

In evaluating the RAP, the performance for the improved AU design is used. Within the context of the RAP, extra time is needed to change switch configurations each word time. This results in a word time being extended to 5 major cycles, and each AU having a peak performance of 4MFlops.

### 2.3.2 Switch

The switch topology is shown in Figure 2.5. Each AU selects one of 8 inputs for each of their two operands, while the feedthroughs each have the choice of 4 inputs. On the first configuration of any given method the inputs are taken from the 16 input registers, while on subsequent configurations the inputs are taken from the outputs of the AUs and feedthroughs. The column of 2X1 multiplexers is used to make this choice.

The switch chosen does not offer complete connectivity in which any output can be connected to any input. In fact, it has less than half of the connections of a complete crossbar. The incomplete switch has the advantage of being faster and requiring less state information. It is faster because there is less than half the capacitance on the input and output lines to the switch that there would be in the case of a complete crossbar. The amount of state required to describe each configuration is only 68 bits (3 bits for each AU input, 2 bits for each feedthrough, and 1 bit for each adder/subtractor to select the add or subtract function) which fits into a single 72 bit word. This allows a change of the switch configuration in a single major cycle by reading a single word from memory. For the benchmark problems used to evaluate performance, the incomplete connectivity did not prevent an efficient mapping of the problems onto the switch. The principal reason for this is that

---

what these modifications are.

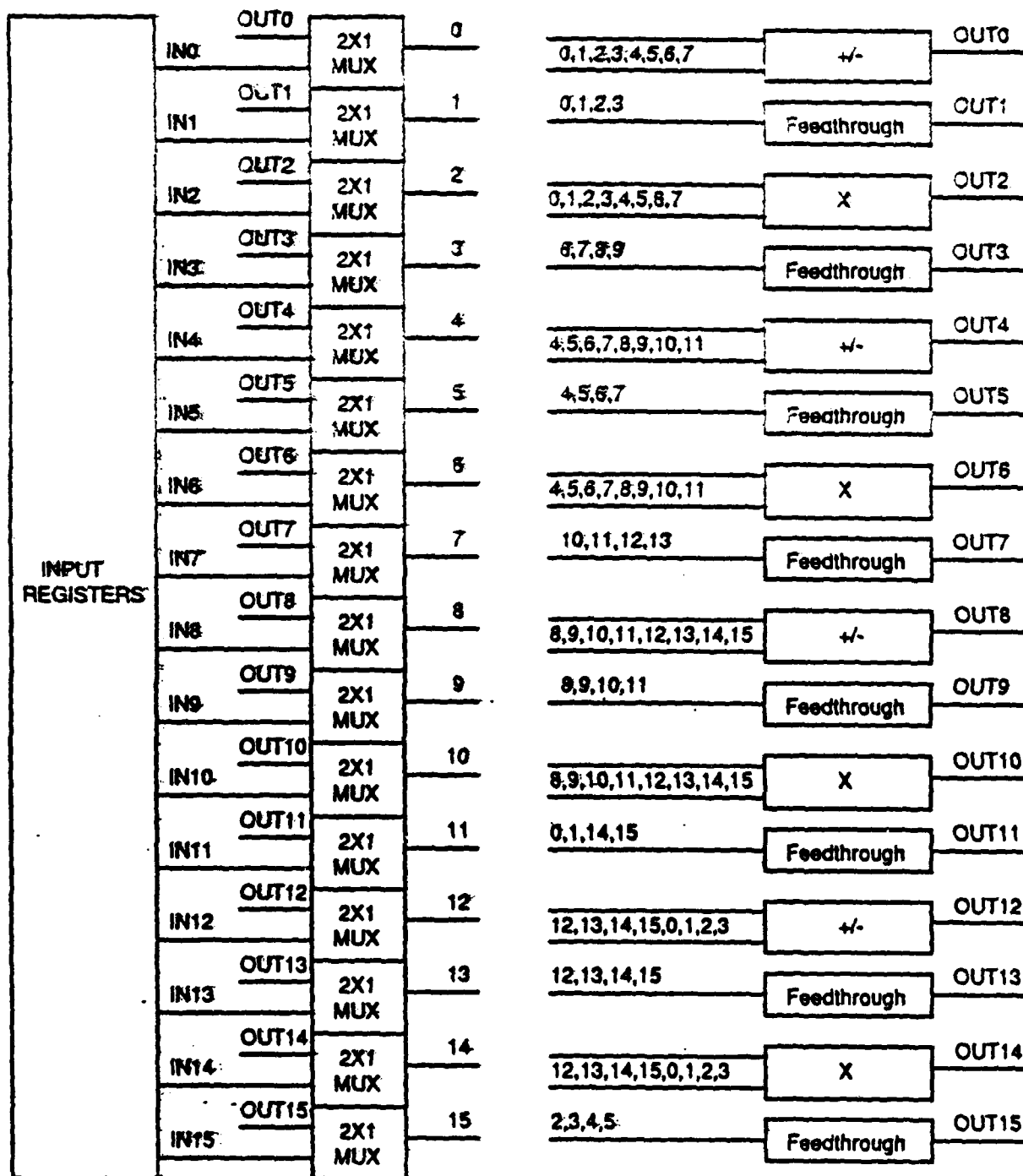


Figure 2.5: Switch Configuration

there are multiple units capable of performing the same function (add/subtract, multiply), so complete connectivity results in unnecessary flexibility: with proper scheduling, it is possible to limit the choice of units to a small subset of the AUs performing that function, without losing much performance. With the switch shown, the RAP achieves 88% of the performance that is achieved if the switch used is complete (see Chapter 6). In Chapter 5 an expression compiler which maps a given equation or set of equations into a series of appropriate switch configurations is described.

### 2.3.3 Using the Datapath

There are several ways that the RAP datapath can be organized in terms of how it calculates expressions. The simplest way, which does not require the AU to allow consecutive problems to be pipelined one directly behind the other, is to have all functional units have the same delay, including feedthroughs. In this case, only one operation can be computing in each functional unit at once. Since the design for the floating-point units in Chapter 4 has a delay of approximately three word times for both the adder/subtractor and the multiplier, having only one problem computing at once means that at any given time 2/3 of the circuitry will be inactive. Figure 2.6 shows how the sum of 8 numbers would be accumulated using two adders and four feedthroughs. In this Figure, a snapshot of the state of the datapath is shown for each word time. In word time one, two adds are initiated in the adders,  $X1 + X2$  and  $X3 + X4$ . Since only two adders are available, the remaining inputs,  $X5$  through  $X8$  are sent to feedthroughs (FT). In word time two, all operations have advanced one word time through the functional units. At the end of word time three, results begin to shift out of the functional units, and two more adds can be initiated in word time four ( $X9 + X10$  and  $X5 + X6$ ). Operation continues until all the terms have been added together.

If the design of the AUs allows the pipelining of consecutive problems, the speed of computation can be increased as shown in Figure 2.7. In this case, a feedthrough is turned into a one word time delay which delays an operand until the other operand is available

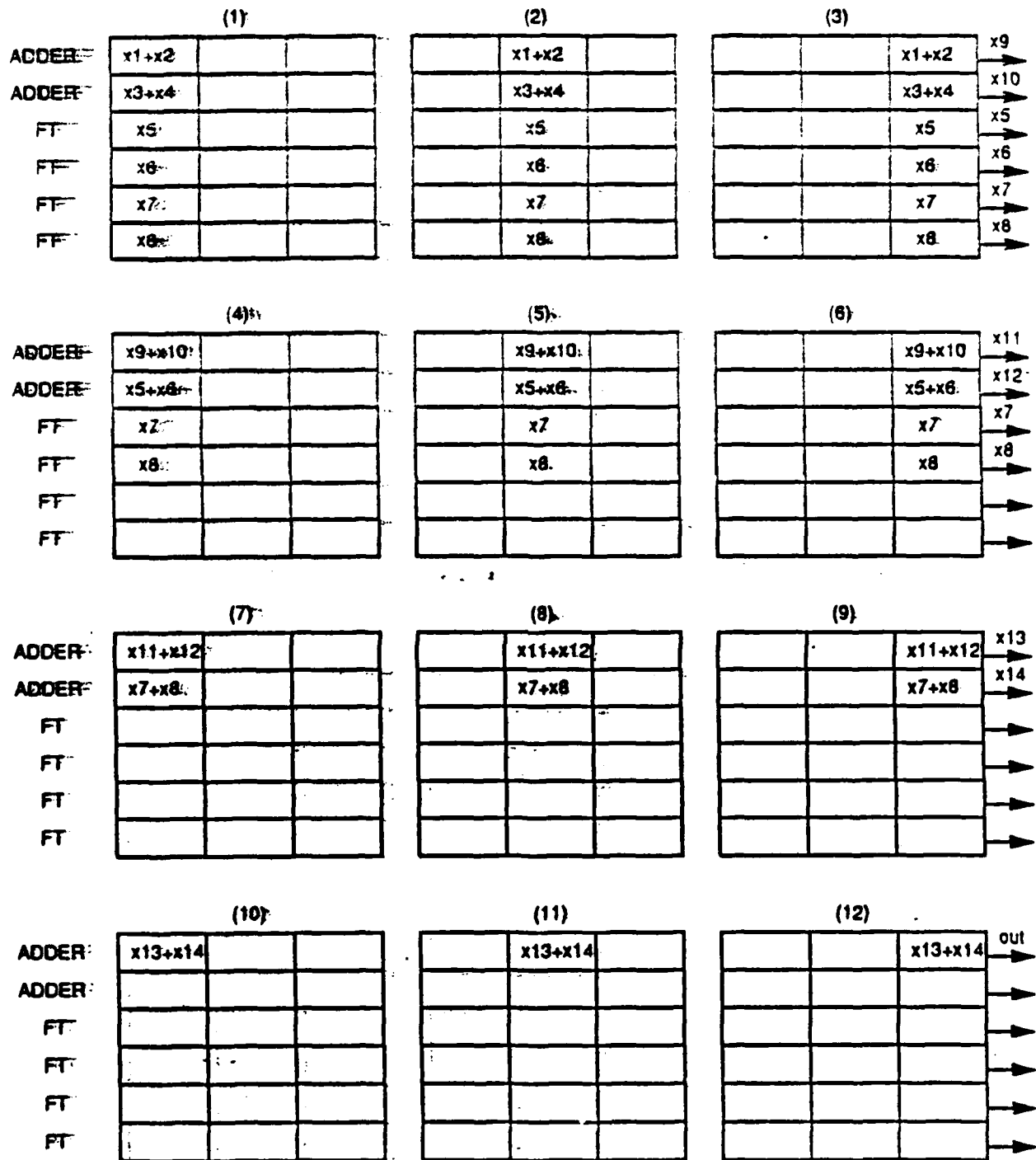


Figure 2.6: Computing a Single Problem Without Pipelining

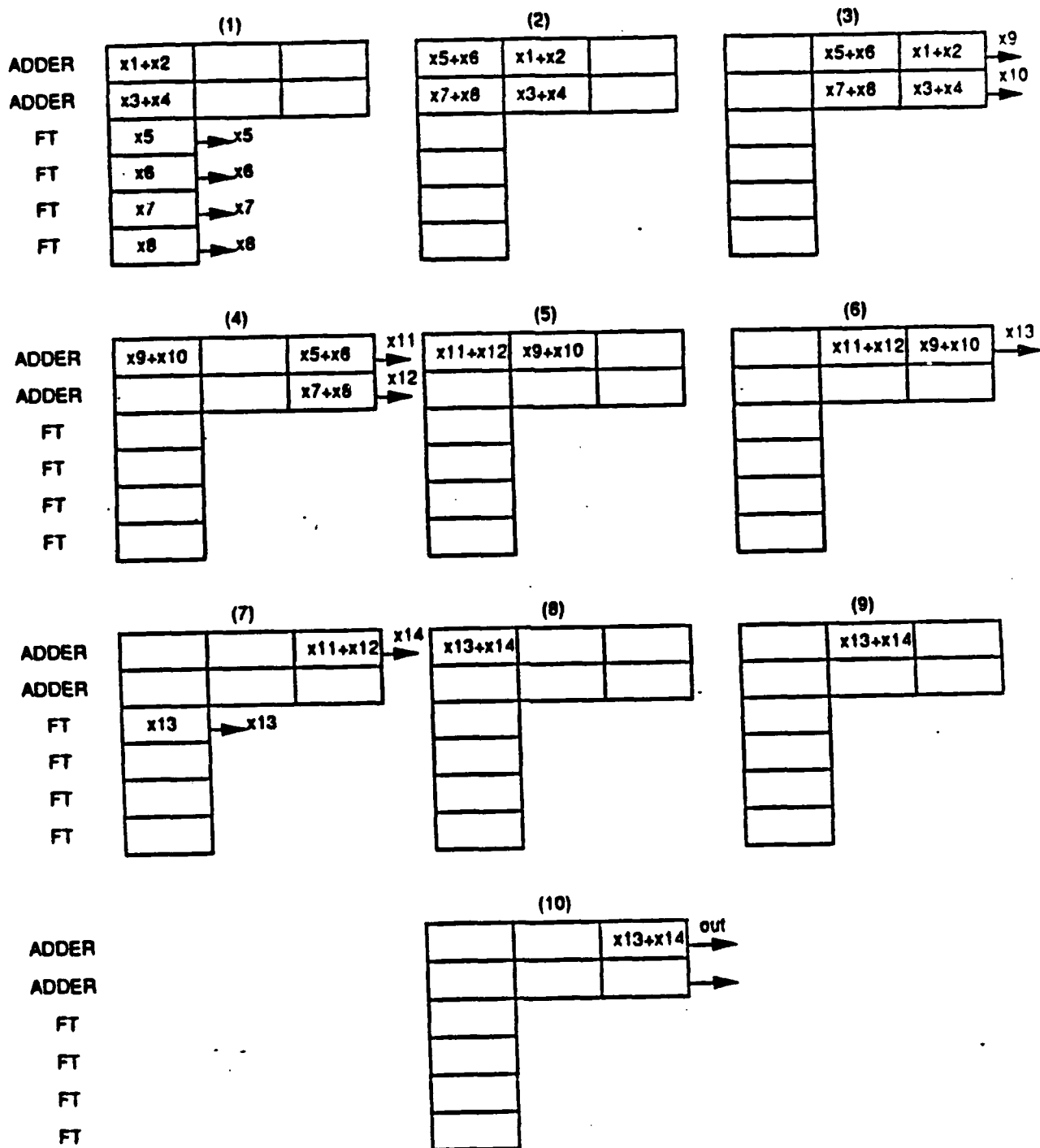


Figure 2.7: Computing a Single Problem Using Pipelining

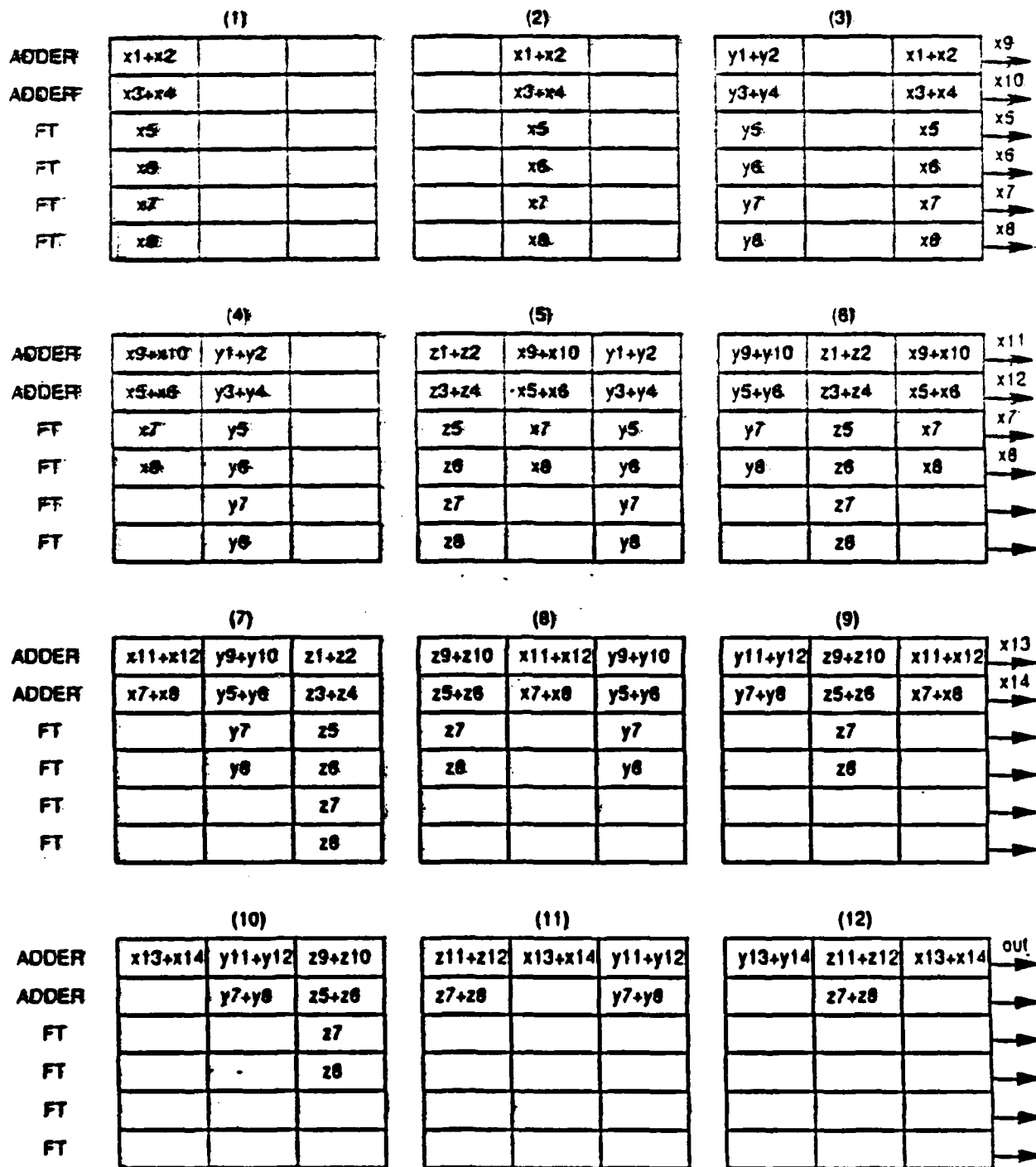


Figure 2.8: Computing Multiple Instances of the Same Problem Using Pipelining

and there is a slot free in the AU pipeline. In word time two of Figure 2.7, two new add operations can be initiated since the adders are available, and operands X5 through X8 are available as outputs from the feedthrough units. Note that this scheme also has the advantage of allowing units with variable delay: an adder with a two word delay could be used with a multiplier with a three word delay without having to insert an extra word delay in the adder.

Alternatively, the pipelined AUs could be used to do multiple instances of the same problem. In order to do multiple instances of the same problem, the datapath is thought of as three different machines doing the same computation (i.e. using the same method) on different operands. This is similar to the multi-threaded execution that occurs in the HEP [34, 17], in which several instruction streams are used to keep a pipeline busy. Figure 2.8 shows an example of this in the case of the 8 number accumulate. In this example, as many as three different problems are active in the datapath at once. Notice that the empty slots in the datapath do not necessarily get filled right away because time is needed to load the next set of operands into the input registers. This example assumes that a new set of operands can be loaded and ready to go in two word times.

There are extra costs associated with both ways of exploiting the AU pipelining. Using pipelining to increase the speed of a single computation requires a new switch configuration every word time. This considerably increases the number of switch configurations in a method and the storage required for each method. In the example given, 4 switch configurations were required in the method in the non-pipelined case, as compared to 10 in the pipelined case. In some cases this method does not succeed in reducing the time spent in doing a calculation (consider the case of accumulating 6 instead of 8 numbers in a sum), but requires 3 times the number of switch configurations in the method definition<sup>3</sup>. Having

---

<sup>3</sup>It is possible to find a way of compacting the method definition in this case. For instance, a word could be added to the method definition in which each bit of the word indicates for each word time of the method whether the switch configuration should change, or whether it should remain the same as in the previous word time.

several different problems in the datapath requires hardware to keep track of where each of them is in the method (i.e. it is necessary to keep three different "Instruction Pointers" into the current method). Also the switch configuration may have to be changed three times as often (once every word time) as the non-pipelined case. Furthermore, in order to exploit this type of usage, the problem being solved must require that the same method be calculated many times with different operands.

All things considered, the best way of exploiting the pipelining of problems in the AUs is to use pipelining to increase the performance of a single problem instance. No extra control hardware is required over the case where the AUs are not pipelined, and it is not necessary to have many instance of the same problem in order to achieve good performance.

## 2.4: Comparison to Another Approach

A more conventional way of dealing with the off chip I/O bandwidth problem is to include a register file on chip as shown in Figure 2.9. Intermediate results are stored in the register file so that they can be reused without going off chip. This reduces the off chip bandwidth in the same way that the RAP does, by keeping intermediate results on chip. Having a register file on chip moves the I/O problem on chip where it can be dealt with more easily by using multiple ports. Using multi-ported register files to solve the off chip I/O problem does not use chip area as efficiently as the RAP, and is more difficult to control, as is discussed in the following sections.

### 2.4.1 Parallel Arithmetic vs. Serial Arithmetic

Serial arithmetic uses chip area more efficiently than a parallel combinational approach. For purposes of this section, efficiency is defined to be  $P/A$  where  $P$  is the performance achieved and  $A$  is the cost of this performance in terms of the chip area. Figure 2.10 compares the

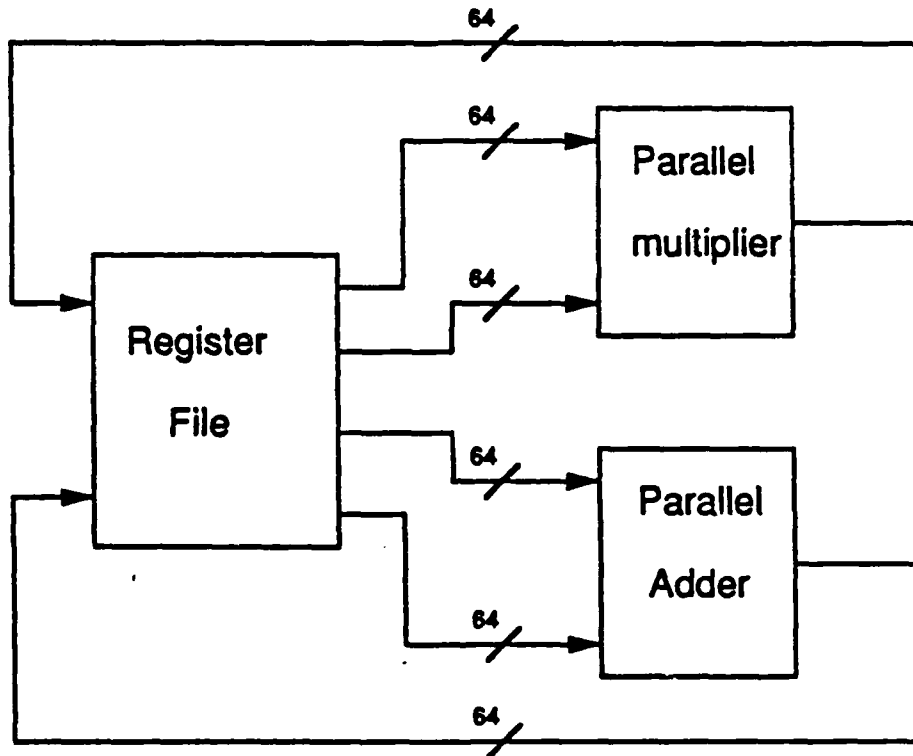


Figure 2.9: Parallel Arithmetic Using a Register File

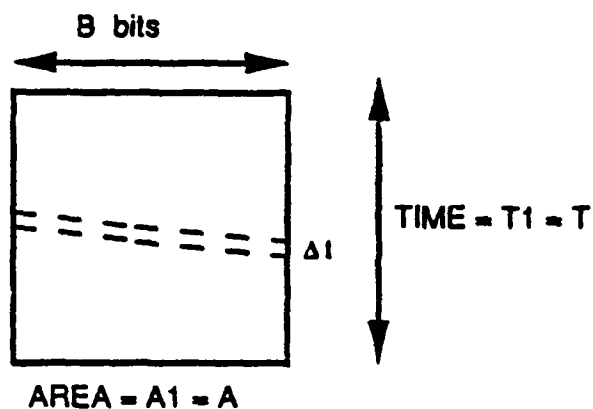
relative efficiency of combinational, parallel pipelined, and serial pipelined approaches.

As a starting point consider the combinational case. Performance  $P$  is defined as  $B/T$  where  $B$  is the number of bits done in parallel, and  $T$  is the time required to do the computation. The area required for the implementation is  $A$ , so that the efficiency of the implementation is just  $P/A$ . In the combinational case only a small portion of the logic is active at any given time, corresponding a wavefront of activity of width  $\Delta t$  propagating through the circuit, where  $\Delta t$  is the switching time of the circuitry.

When the combinational unit is pipelined by dividing it into  $n$  stages and inserting latches between the different stages, there is an increase of performance because there are  $n$  wavefronts of computation active at the same time. This increase in performance is not without cost: there is a percentage increase  $((n - 1)\epsilon)$  in the area required due to the increased circuitry, and a percentage increase  $((n - 1)\delta)$  in the time of computation due to synchronization costs and delay of the latches. These increases are proportional to the number of latch stages added. The net result is that pipelining increases efficiency up until the point that the cost of the latches becomes significant. In order to get some feeling for the cost of the latches, reasonable values for the parameters  $\epsilon$  and  $\delta$  can be derived from the arithmetic cell designed for the prototype fixed point RAP, which is described in more detail in Chapter 4. This cell does two bit arithmetic and requires  $B/2$  stages where  $B$  is the number of bits. The area of the cell is  $9.7K\lambda^2$  and the latches require approximately 50% of the area and contribute approximately 50% of the delay. Using the expression for the area in Figure 2.10b the formula  $A_2 = 2A = A + (B/2 - 1)\epsilon A$  must hold, from which it is determined that  $\epsilon = 2/(B - 2)$ , or 0.032 when  $B = 64$ . Making the conservative assumption that the time in the circuitry increases linearly with the number of bits, and also that the time spent in the latches is constant, then the percentage cost in time for each stage is  $\delta = 2/B$ , or 0.031 when  $B = 64$ . For these values of  $\epsilon$  and  $\delta$  maximum efficiency is achieved when  $n = 30$ .

Comparing the serial case with the parallel case is complicated because their modes of

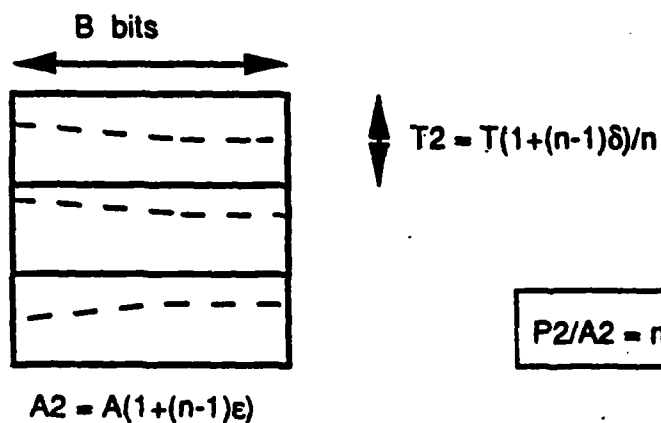
1) COMBINATIONAL



$$\text{PERFORMANCE} = P_1 = P = B/T$$

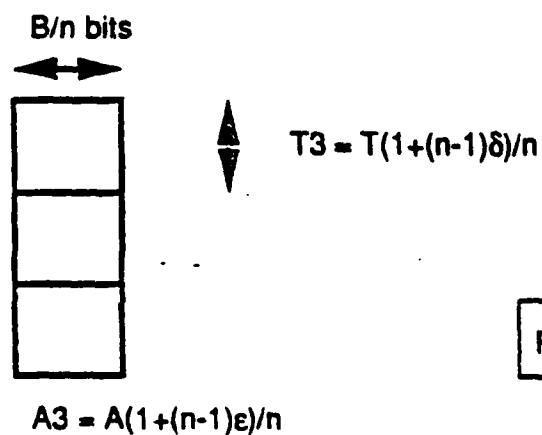
$$P_1/A_1 = P/A$$

2) PIPELINED (n STAGES)



$$P_2/A_2 = nP/(A(1+(n-1)\epsilon)(1+(n-1)\delta))$$

3) SERIAL



$$P_3/A_3 = nP/(A(1+(n-1)\epsilon)(1+(n-1)\delta))$$

Figure 2.10: Parallel Arithmetic vs. Serial Arithmetic Efficiency

operations are so different. In the pipeline case each stage completes its work on a given problem and the next inputs are from another problem. In the serial case each stage works on the same problem several cycles, receiving part of the input at each cycle. However, by making a number of simplifying assumptions, the serial case can be compared to the parallel case:

1. Assume that the number of serial stages is inversely proportional to the number of bits done in parallel.
2. Assume that the time required for a given stage is directly proportional to the number of bits done in parallel.

The first assumption implies that if doing one bit at a time requires 64 stages, then doing 2 bits at a time would require 32 stages. The area of multiplier array structures as well as of a number of other structures normally used in floating-point circuits (e.g. barrel shifters), scales as  $(B/n)^2$  where  $(B/n)$  is the number of bits done in parallel. Reducing the number of bits done in parallel by the factor  $n$  would normally reduce the area by a factor of  $n^2$ . However, under the above assumption the area is reduced only by a factor of  $n$ , since reducing the number of bits done in parallel increases the number of stages. This assumption is reasonable for most algorithms but it does not take into account algorithms which require only a fixed number of stages independent of the number of bits done in parallel (examples of these are integer add which requires only a single stage, and some of the digit-on-line [16, 35, 29] class of algorithms). Practical floating-point algorithms are difficult to implement in this way because of the mutual dependence of the exponent and the mantissa fields.

The second assumption is a pessimistic assumption since time can be made to scale as  $\log(B/n)$  for many operations. This assumption is sufficient for a first order approximation since  $(B/n)$  is no larger than 64 for the cases of interest, and since achieving the logarithmic time usually involves a corresponding increase in the area costs.

Under these simplifying assumptions, the expression for the efficiency in the serial case shown in Figure 2.10 is the same as for the heavily pipelined case. The big advantage of the serial implementation is that it achieves an incremental extensibility not achievable in the parallel case: if there is enough bandwidth to maintain 8MFlops of computation, then two 4Mflop serial units can be used without paying the area costs of a full parallel implementation. Thus the serial implementation provides both efficient use of the silicon area and allows an incremental increasing of the computing power. The disadvantage of serial arithmetic is that it cannot use logarithmic algorithms that can be used in parallel implementations (e.g. Wallace Tree multipliers [18]), which increase the speed of a single operation.

#### 2.4.2 Register File vs. Switch

The use of a switch rather than a register file is more efficient in terms of area and is easier to control. Figure 2.11 shows the area required for both options. The register file has fewer ports but requires that each port be the full 64 bits wide. The switch has a larger fan in and fan out but uses narrow serial data paths. For relatively small switching requirements, the switch is more area efficient. For sake of comparison, compare the Weitek 3164/3364 [4] chip which has a peak performance of 20MFlops, to the RAP which has a potential peak performance of 32MFlops. The 3164/3364 has a 32X64 register file with 6 input and output ports, which requires 6 times the area of the RAP switch which routes data 4 bits at a time, has 32 input ports, and 24 output ports.

On a more qualitative level, general routing in the case of the register file is made difficult by the need to route multiple full width busses. In order to solve the I/O problem the register file must have multiple ports (6 ports are used in the Weitek 3164/3364) and the area required to do general routing of these busses is proportional to  $PB^2$  where  $P$  is the number of ports and  $B$  is the number of bits. Routing in the serial case is all done within the switch. The above area comparison neglects a number of the extra costs associated

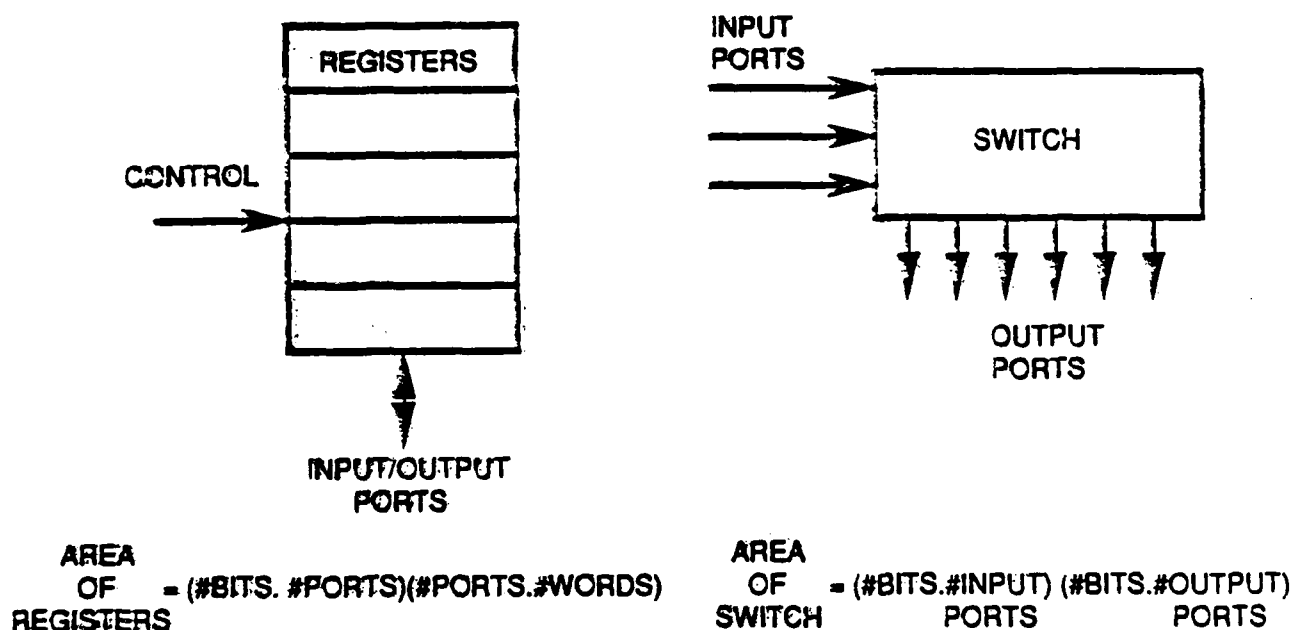


Figure 2.11: Register File Area vs. Switch Area

with each of the methods: the register file requires address and decoding logic for each of its ports and requires bus multiplexing into the functional units, while the switch requires switch control logic and parallel-serial/serial-parallel conversion registers.

Control of the switch is easier than control of the register file because control signals have to be changed only once every word cycle, whereas control to the register file must be available every clock cycle. In the case of the switch, the control signals remain the same for at least 16 minor clock cycles as data propagates serially through the switch. Delay paths through the switch can be optimized without worrying about control overhead. In the case of the register file, there is a control cost associated with every clock cycle since at each cycle the register addresses of source and destination registers must be provided and decoded.

## 2.5 Summary

The RAP architecture uses serial floating-point arithmetic units in combination with a flexible switch to route data between units. This scheme can lead to a substantial reduction in the I/O bandwidth required to sustain a given level of computation. It does this by eliminating all bandwidth costs associated with storing and retrieving intermediate results.

The RAP is designed with a message passing interface so that it fits into the J-Machine, a message passing concurrent computer. The RAP is controlled by three messages which allow it to store "methods" and forwarding information in its local memory, and which allow it to execute these methods on different sets of data. Within this system, RAPs can be set up in a RAP pipelines, fork operations can distribute work over several RAPs, and merge operations can combine the results from several RAPs.

The block diagram of the RAP includes the control blocks, memories, and the datapath. The control blocks are set up so that the operations of receiving a message, loading operands into the datapath, computing a problem instance, unloading output results, and sending result messages can all be pipelined. The main RAP memory is used to store methods and result forwarding information. Input and output queues are used to buffer incoming and outgoing messages respectively. The datapath contains the 16 functional units, including 4 add/subtract units, 4 multiply units, and 8 feedthrough units. It also contains the switch that routes the data between units, input registers, and output registers.

The use of bit-serial arithmetic and a switch, as in the RAP, can be compared to the use of bit-parallel arithmetic and a register file, as in more conventional approaches. the serial/switch approach uses logic efficiently, while using less area, and being easier to control than the parallel/registers approach. The next two Chapters will concentrate on two important aspects of the RAP design: the control logic and the datapath design.

## Chapter 3

# RAP Simulation and Control Logic

*I claim not to have controlled events, but  
confess plainly that events have controlled me.*

— ABRAHAM LINCOLN in Letter to A.G. Hodges, April 4, 1864

*She makes me wash, she comb me all to thunder... The  
widder eats by a bell, she goes to bed by a bell; she gits up  
by a bell - everything's so awful reg'lar a body can't stand it.*

— MARK TWAIN in *The Adventures of Tom Sawyer* (1876)

In order to design and debug the control logic, and to evaluate performance, a register transfer level simulator was written to simulate the RAP. The simulator is written in LUCID Common LISP and runs on the SUN workstations. This chapter describes the simulator organization and presents the flow charts for the control logic of the RAP chip.

### 3.1 Simulator Description

The simulator is a discreet event simulator: events are scheduled on a priority queue, and at each tick of the clock events scheduled for that time are taken from the queue and executed, causing other events to be scheduled. The simulator was written in an object oriented fashion using the LUCID LISP flavors package [22]. Several RAPs can be simulated at once. Commands are provided to set up the simulation, run the simulation, and observe the state of the RAPs as the simulation progresses.

A number of simplifying assumptions were made. For instance, memory for methods was allocated statically in fixed size blocks. In the real RAP memory management will be done more efficiently by storing methods in variable sized blocks. Another simplification is that the simulator knows nothing about the switch topology: it simply executes the given operations. Switch constraints must be enforced by the user or compiler that generates the sequence of switch configurations in the method. These simplifications do not affect the performance results obtained.

#### 3.1.1 Simulator Organization

The simulator code is divided into the following main components:

1. **Global variables and Constants.** These include such things as the current time step, the locations being watched at each simulation step, and the size of the RAP memory and queues. They also include simulator parameters which represent the delay associated with certain events such as reconfiguring the switch, or a word arriving at the network interface.
2. **Definition of a RAP object type.** The RAP is described as an object consisting of all the sub-blocks of the RAP. Each sub-block of the RAP in turn is also described

as an object. The major components of the RAP object are the input and output queues, the memory, the datapath, the network, the control, and the priority queue of pending events.

3. **Internal Event Control Programs.** The input control, output control, switch control, and network interface control, are all described as RAP methods. They explicitly describe the RAP's internal control. Separating the control programs of the RAP means that the control algorithms are directly implemented and they have a straightforward translation to hardware.
4. **External Event Control Programs.** RAP methods are written in order to control events which are not directly controlled by the internal control of the RAP, such as words arriving at or leaving from the network interface.
5. **Priority Queue Manipulation.** Procedures used to manipulate the priority queue of events. The queue is just an ordered list.
6. **Command Programs.** Programs written to set up a simulation, to run the simulator, to look at RAP state, and to perform other useful functions for debugging and observing results.

## 3.2 Control

The control of the RAP is divided into four independent sections: input control, output control, switch control, and network interface control. Input control executes incoming messages, and controls the input to the datapath, and most memory operations. Output control is responsible for constructing result messages in the output queue, while switch control is responsible for loading switch configurations at the correct time. The network interface logic controls how words coming from the network are put into the input queue and how words are sent out to the network from the output queue. These control blocks

must communicate to synchronize the different stages of the calculation, and to prevent the queues from overflowing.

In order to implement the control, various registers and signals were introduced to interface with the memory, the network, and the datapath. These control registers and signals are listed below. These list should be used as references when going through the control flow charts which follow.

The control registers are:

1. **Queue registers:** HEAD and TAIL registers for both the input and the output queues, referred to as HEADIN, TAILIN, HEADOUT, and TAILOUT.
2. **Memory registers:** Four registers (A0-A3) used to read and write memory.
3. **Datapath Input/Output control registers:** Six bit-vector registers used to control the loading and unloading of the input and output operand registers. These registers contain 1 bit for each input or output register.
  - (a) **AIRREF, AIRCNT:** Address of Input Registers Reference and Count registers. These registers are used to control the loading of the input registers. AIRREF identifies which input registers should be loaded for the method currently executing. AIRCNT points at the next input register to be loaded.
  - (b) **AOREF, AORCNT1, AORCNT2, AORCNT3:** Address of Output Registers Reference and Count registers. These registers are used to control the unloading of the output registers. At any given time there may be four problems in the datapath: one being loaded into the input registers, one being being calculated by the AUs, one being shifted out of the AUs into the serial part of the output registers, and one being unloaded from the parallel part of the output registers into the output queue. Each of these four registers corresponds to one of the four possible problems in the datapath. The three AORCNT registers also have an

extra bit which indicates whether the problem is the last problem of the message. AORCNT3 is used to point at the successive output registers as they are being unloaded.

4. **Configuration control registers:** These registers control the switch and the sequence of configurations it runs through.

- (a) **SWITCH:** register used to store the current switch configuration.
- (b) **CRREF1, CRREF2, CRCNT:** Configuration Register Reference and Count registers. These registers are used to control the loading of switch configurations. CRREF1 contains the number of configurations in the method which will be executing next, CRREF2 contains the number of configurations of the method which is currently executing, and CRCNT keeps track of which configuration in the method is currently loaded.

5. **Network Interface Registers:** These registers are used to send and receive words from the network.

- (a) **NET-IN-REG:** Register used to receive a word from the network.
- (b) **NET-OUT-REG:** Register used to send a word to the network.

6. **QIN-REG:** Register in which the word dequeued from the input queue is stored.

The control signals used to communicate between the different control blocks are:

- 1. **create-template:** indicates whether the input control can load a new forwarding template into the template buffer. Set by the output control, reset by the input control.
- 2. **go-download:** indicates whether the next problem can be shifted into the AUs. Set when a download is allowed to occur, reset by the input control on a download.

3. **go-unload**: indicates whether the output registers are ready to be unloaded. Set when the results of a calculation are uploaded into the parallel part of the output registers, reset by the output control.
4. **reconfigure**: indicates that the switch should be loaded with the next configuration. Set internally a predetermined amount of time after a new problem has been downloaded or after the switch has just been reconfigured, reset by the switch control logic.
5. **template0**: indicates whether the default template is being used. Set and reset by the input control.
6. **net-in-status**: indicates whether the network has a word ready to be enqueued. Set by the input network logic, reset by the input enqueueing logic.
7. **net-out-status**: indicates whether the network is ready to receive an output word. Set by the output network, reset by the the output dequeuing logic.
8. **end-count**: counts the number of complete messages that are in the output queue. Incremented every time the last word of a message is inserted into the queue, decremented every time the last word of a message is sent out to the network.
9. **started-before-end**: indicates that message transmission began because the output queue filled up rather than because a complete message was ready to transmit. Set and reset by the network output control.

### 3.2.1 Input Control

The input control flowchart is shown in Figures 3.1 through 3.3. It is important to note that throughout this flowchart there are two types of implicit wait states. The first is reading the input queue, which may require waiting until a word is present in the queue. The second is reading memory, which may require a delay of one cycle if the switch control is

reading memory. The memory has an input port and an output port so that no waiting is required when writing memory, although this could easily be modified to assume a single port memory.

Initially the input control looks at the first word of a message and begins a control sequence based on the type of message. If the message is a SM or ST message, then the method or template is simply stored in memory. In the case of a C+E the control sequence is more complicated: first the various registers are set up to prepare execution of the method, the NODE ID/REPLY ID is inserted into the template slot reserved for it in memory, and the first set of operands are loaded into the input registers. At this point, the control must make sure that no interference will occur with the previous problem. If the previous message execution has finished with the three words of template buffering (shown as *TEMPL1*, *TEMPL2*, and *TEMPL3*), then the input control loads the new template into the template buffer locations. Then, if the previous message is finished with the datapath, the input control sets up the switch sequencing registers (*A2*, *A3*, *CRREF2*, *CRCNT*) and begins the calculation.

The input control continues loading and calculating problem instances until the end of message is reached. The input control only allows a download to occur if the *go-download* signal is asserted. The *go-download* signal is only asserted if the previous problem has run through its last configuration and will have sufficient time to unload its results, and there is sufficient room in the output queue to hold the results (the *go-download* signal is discussed in more detail in the switch control section).

The input control logic transmits information to the output control logic as to which registers contain results, using the *AORCNT1*, *AORCNT2*, and *AORCNT3* registers. The input control only has to load the *AORCNT1* register and this value gets shifted automatically from *AORCNT1* to *AORCNT2* following the last configuration, and from *AORCNT2* to *AORCNT3* once the results are uploaded into the parallel portion of the output registers. *AORCNT3* is used by the output control logic.

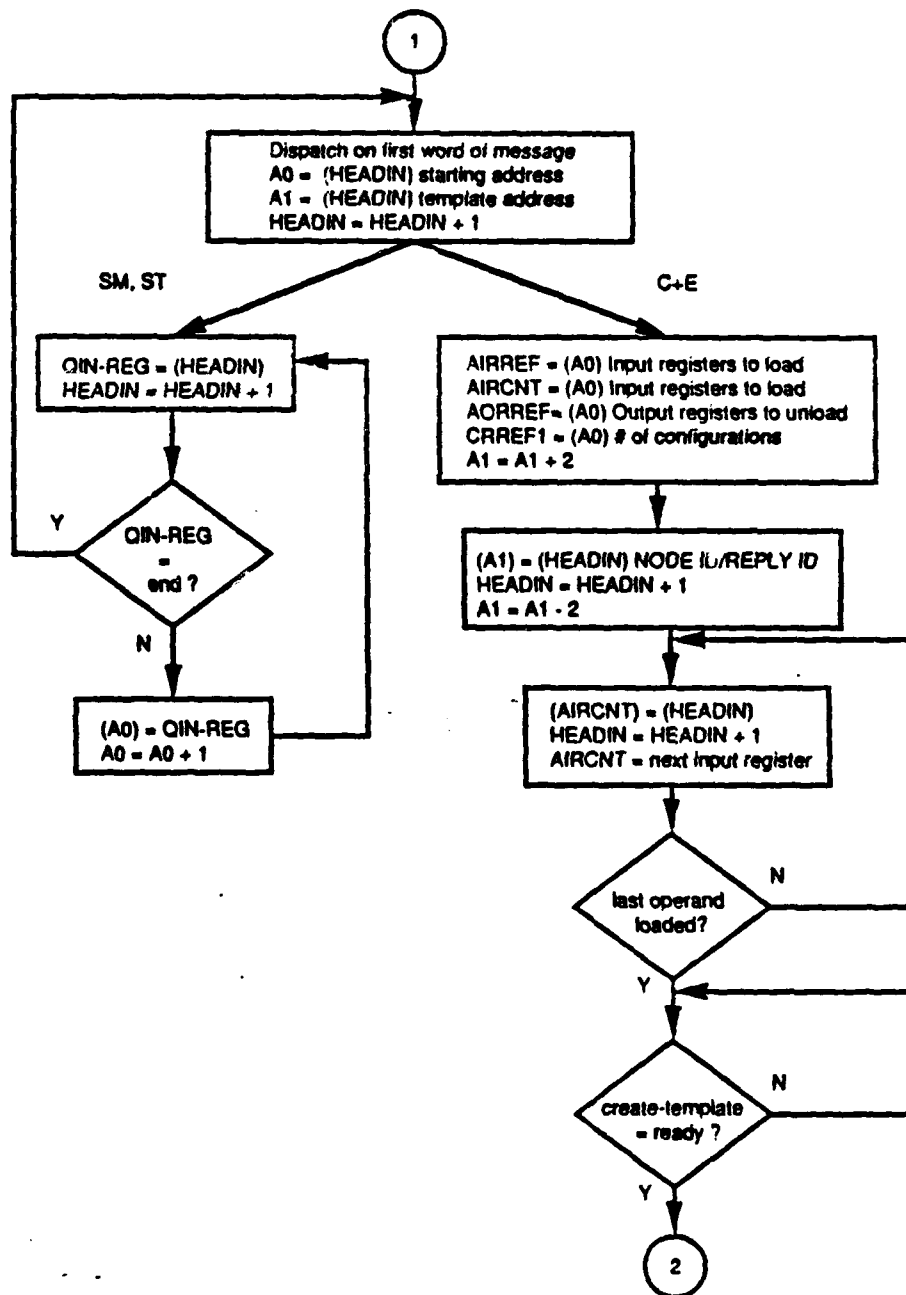


Figure 3.1: Input Control Flow Chart (part 1)

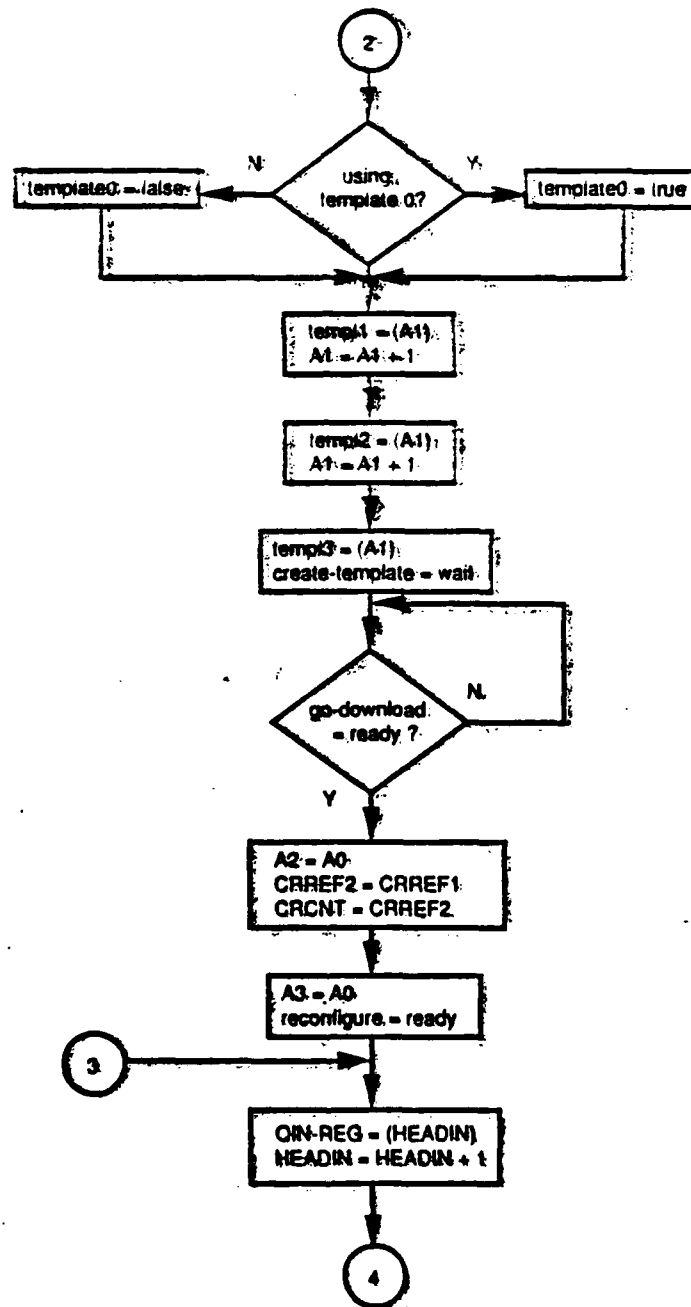


Figure 3.2: Input Control Flow Chart (part 2)

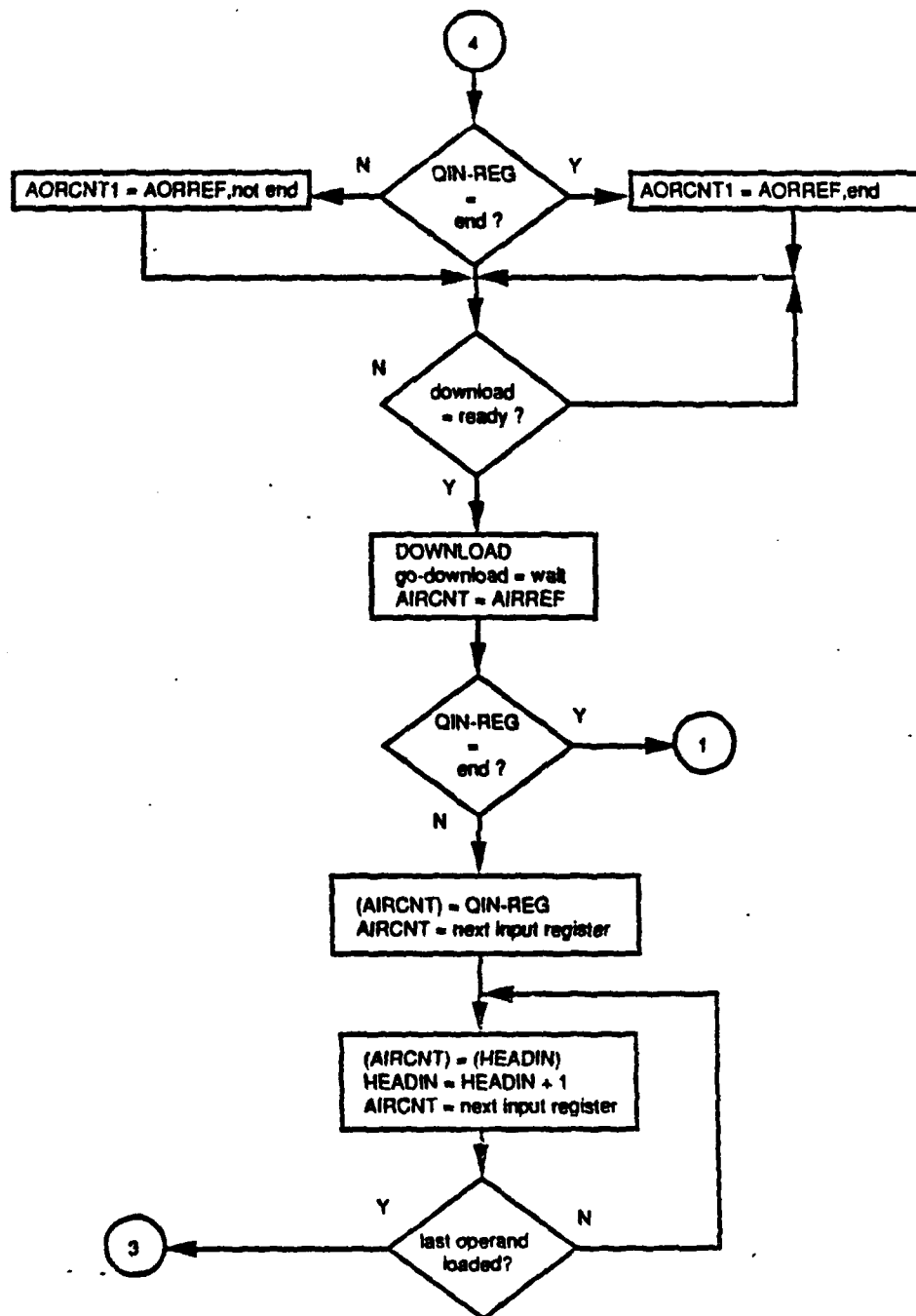


Figure 3.3: Input Control Flow Chart (part 3)

### 3.2.2 Output Control

The output control flow chart is shown in Figure 3.4. The output control is responsible for loading the result messages into the output queue. First the template is output which contains the necessary message header information. If the results are to be sent to their final destination, the *templated* signal is active and only one word of the stored template is needed. Otherwise, all three words of the template are needed to forward the results to another RAP for further computation. The output control then unloads the output registers after the completion of each problem in the message. Once all the results of the message being processed are unloaded into the output queue, the "end" word is appended to the queue to terminate the message. Note that a problem never begins calculating unless it is guaranteed that there is enough space in the output queue to store its output results. This means that the output control will never stall in the middle of unloading the output registers.

### 3.2.3 Switch Control

The switch control flow chart is shown in Figure 3.5. Every time the switch has to be reconfigured the switch control loads the next configuration into the switch register. In the case that a new problem is starting and the first configuration has been loaded, then the switch logic also determines whether the new problem can be downloaded. The problem is allowed to be downloaded if two conditions are satisfied:

1. There is enough room in the output queue to store all the results that will be generated by the computation.
2. Starting the new problem will not cause any results to be lost from problems currently in the datapath. This could occur for instance when the previous problem does not have time to finish unloading the output registers before the results from the next

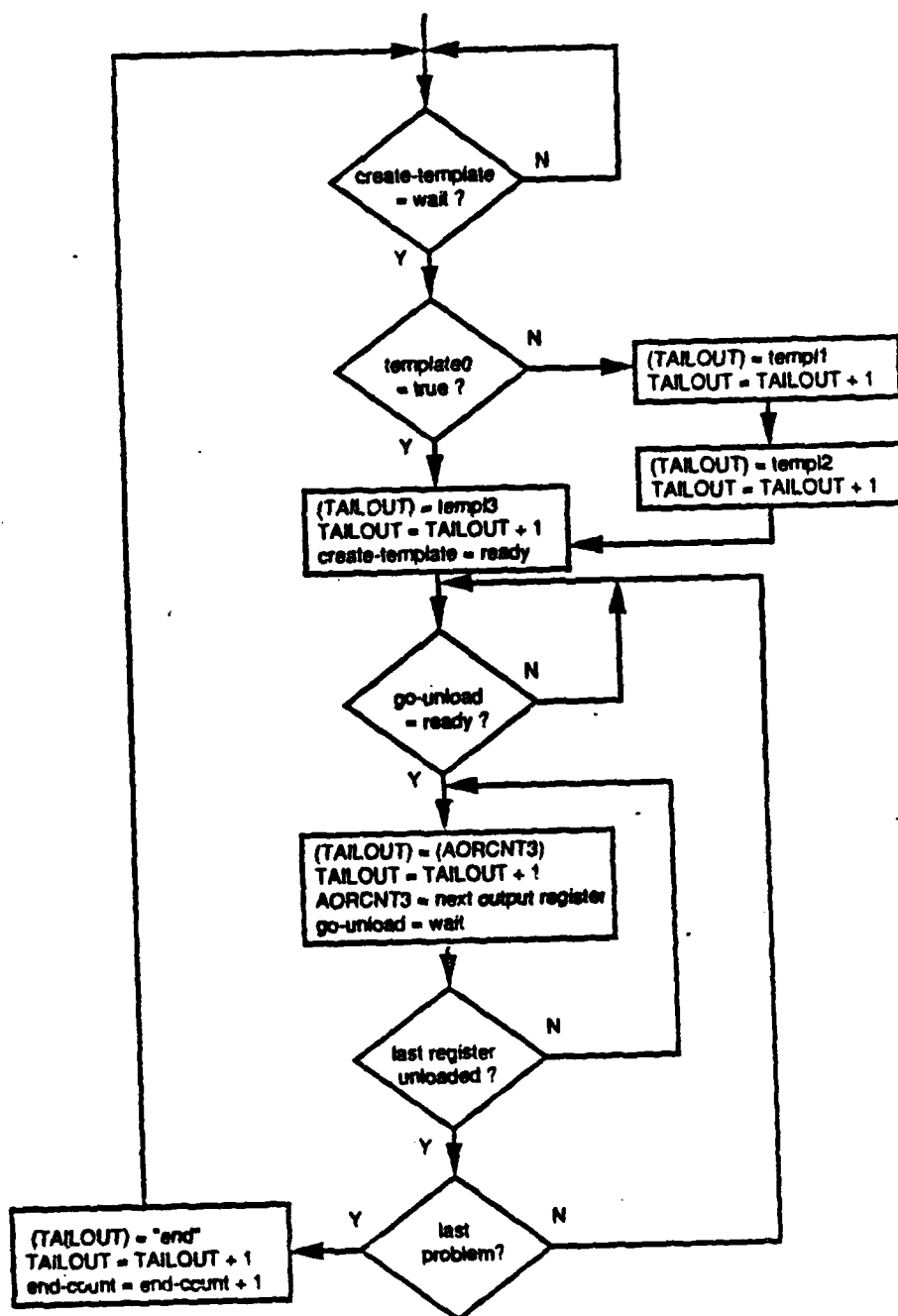


Figure 3.4: Output Control Flow Chart

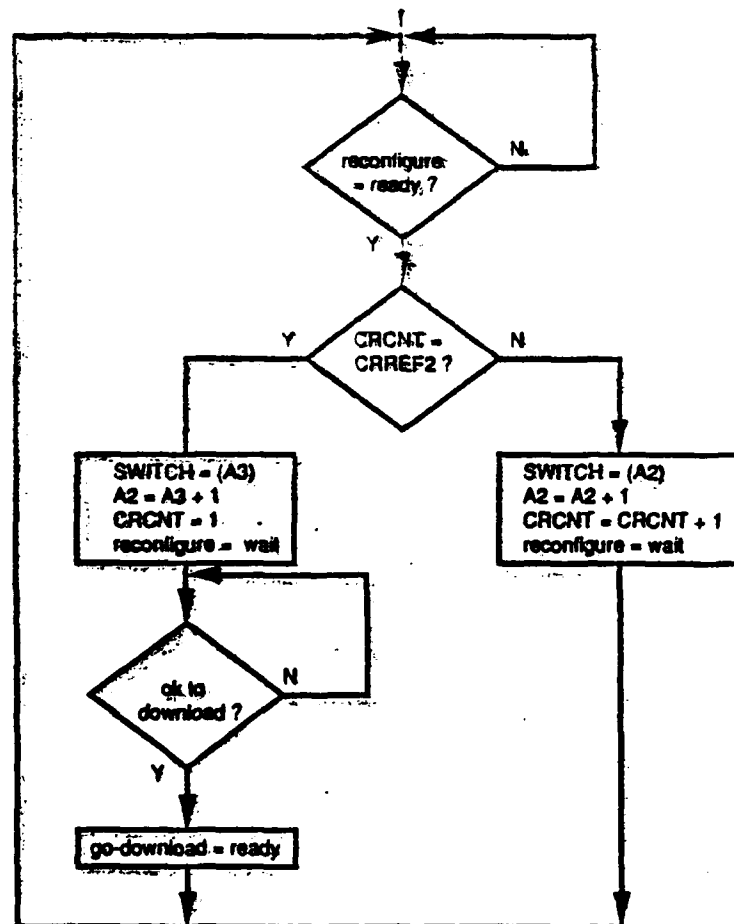


Figure 3.5: Switch Control Flow Chart

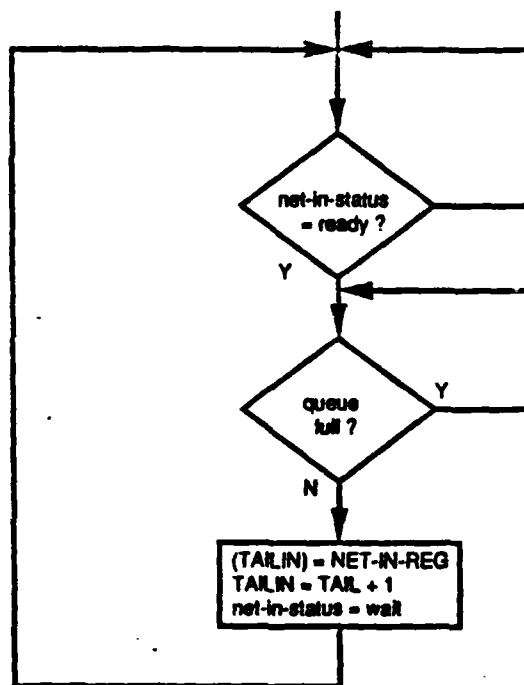


Figure 3.6: Network Input Control Flow Chart

problem arrive. This difficulty is avoided by requiring that each method be at least four configurations, guaranteeing that there will be at least 16 clock cycles to unload the output registers. This is not a costly solution since any method which does useful calculations must be at least three configurations anyway, the number required to do one add/subtract or multiply.

### 3.2.4 Network Control

The network control consists in the interface logic between the input queue and the network and the output queue and the network. Flow charts are shown in Figures 3.6 and 3.7 for the network input logic and the network output logic respectively. The network input logic fills the input queue by taking a word from the network each time a word is ready and

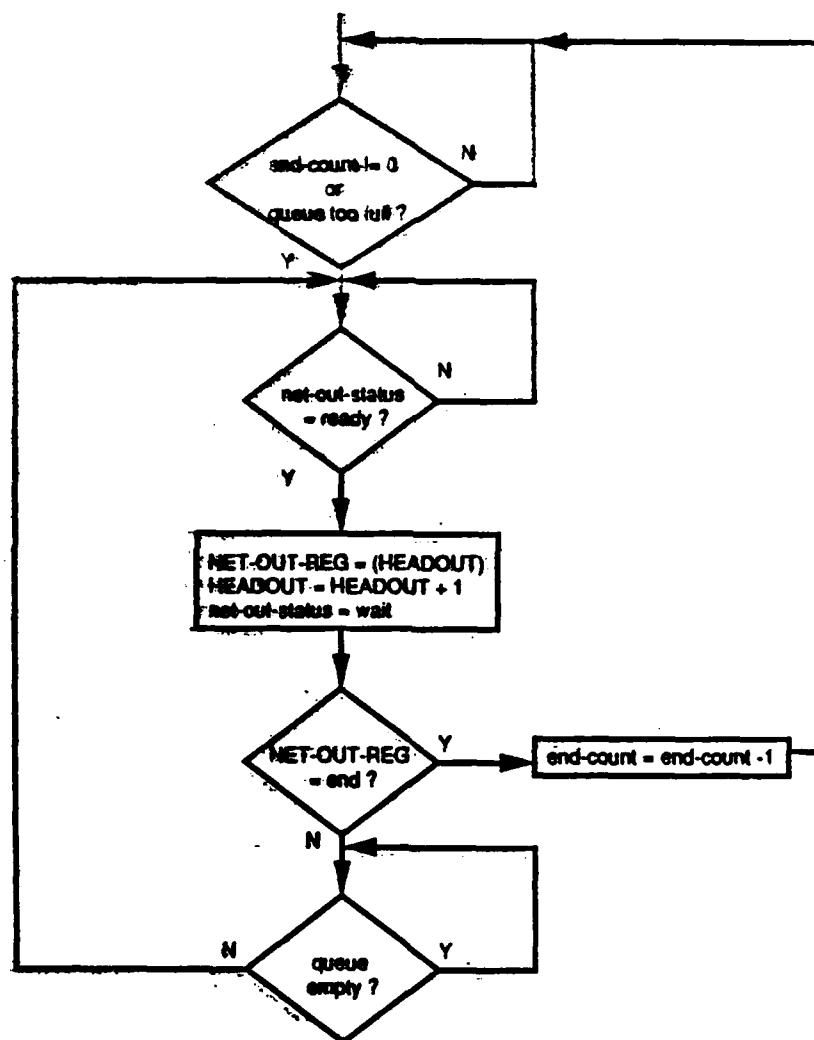


Figure 3.7: Network Output Control Flow Chart

there is room in the circular queue. The network output logic supplies output words to the network beginning when a complete message has been accumulated in the queue or the queue is getting too full, whichever comes first. The number of complete messages in the queue is kept in the *end-count* variable.

### 3.3 Summary

This Chapter presented the flow charts for the control logic of the RAP, including the input, output, switch, and network interface control logic. Registers and signals needed to control the RAP are defined. This logic was verified using a RAP register transfer level simulator and it is straightforward to translate the control flow charts into logic circuitry in the form of random logic or small PLAs.

## Chapter 4

# Hardware Design

*Thunder is good, thunder is impressive;  
but it is lightning that does the work.*

— MARK TWAIN, in a Letter to an Unidentified Person (1908)

*Damn the torpedoes - full speed ahead!*

— DAVID GLASGOW FARRAGUT, at the battle of Mobile Bay  
August 5, 1864.

This chapter discusses the hardware design of the RAP, concentrating on the design of the floating-point units. The hardware design can be divided into different distinct parts: the datapath, the control, the on-chip memories, and the network interface. The most critical portion of the design in terms of proving the feasibility of the RAP is the datapath, containing the floating-point units and the switch. The floating-point units must achieve the target of an 80MHz clock, and be small enough so that several serial units can fit on a single chip. The switch must be able to feed data through at the same speed at which the arithmetic units are being clocked. The other portions of the design such as the memory

and control blocks are straightforward to implement, and the network interface is similar to the design in the MDP.

The floating-point rate achieved by the adder/subtractor and the multiplier in the current design is 1.57MFlops per unit with area estimated to be  $3.2M\lambda^2$  for the add/subtract unit and  $5.6M\lambda^2$  for the multiply unit. The design could be modified to be more highly pipelined resulting in an increased performance of 4.70MFlops per unit, without significantly increasing area. When used in the RAP, these modified units would run at 4MFlops, because some overhead time is required to change switch configurations. A RAP containing four adders/subtractors and four multipliers has a peak performance of 32MFlops and about  $40M\lambda^2$  of the chip area is taken up by the arithmetic units.

The remainder of this chapter discusses the hardware design of the datapath in detail. Section 4.1 describes the number representation and conventions used. Section 4.2 presents a comparative study of a number of 4-bit serial adders, the most critical component of both the floating-point adder and the floating-point multiplier. Section 4.3 describes the operation of the floating-point adder/subtractor and the floating-point multiplier. The remaining components of the datapath design, including the switch and register design, are presented in section 4.4. Different ways of improving the design of these units is described in 4.5. Finally, section 4.6 briefly presents the fixed-point RAP, a chip designed in a course project to experiment with some of the RAP ideas.

## 4.1 Numbering System

### 4.1.1 Format

For simplicity, the non-standard floating-point format shown in Figure 4.1 was chosen. It consists of an 8 bit, two's complement exponent field, and a 56 bit, two's complement man-

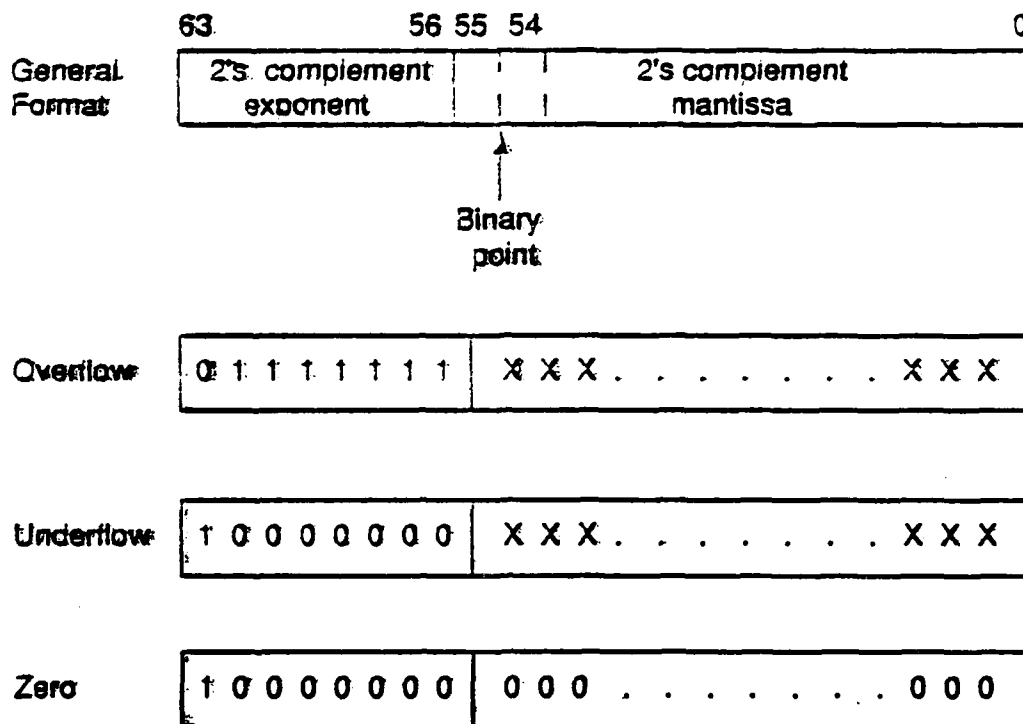


Figure 4.1: Floating-Point Format

tissa field. This format departs from such standards as the IEEE Floating-Point Standard [3, 15] but has the advantage of permitting a uniform treatment of the exponent and mantissa in two's complement form. The exponent is a two's complement number  $E$  in the range  $-128 \leq E \leq 127$ . The mantissa has a binary point following the first bit so that the mantissa  $M$  falls in the range  $-1 \leq M < 1$ . The resulting number is equal to  $M \times 2^E$ .

#### 4.1.2 Normalized Numbers

Only normalized numbers are allowed in our numbering system. A normalized floating-point number is one in which the mantissa  $M$  falls in the range  $-1 \leq M < -1/2$  when  $M$  is negative and in the range  $1/2 \leq M < 1$  when  $M$  is positive. This implies that the second bit of a positive normalized mantissa must always be a 1 and that the second bit of

a negative normalized mantissa will always be a 0 e.g. 01010001011 is a normalized positive mantissa, 00101000101 is an unnormalized positive mantissa, 100111100000 is a negative normalized mantissa, and 111100001100 is a negative unnormalized mantissa. Allowing only normalized numbers as inputs to our floating-point units prevents the unnecessary loss of precision that results from using unnormalized mantissas, and simplifies the logic <sup>1</sup>.

#### 4.1.3 Overflow and Underflow

The result of an operation can fall outside the range of representable normalized numbers, either by requiring an exponent greater than or equal to 127 (overflow) or by requiring an exponent less than or equal to -128 (underflow). Two exponent values are reserved to indicate when either of these conditions occur. The most positive exponent, 127 decimal or 01111111 binary, is used to represent a number which has overflowed. The most negative exponent, -128 decimal or 10000000 binary, is used to represent a number which has underflowed. When an overflowed or underflowed number is used in an operation the conventions shown in table 4.1 are followed.

#### 4.1.4 Zero Representation

It is convenient to have a special representation for zero. Zero is represented by the most negative exponent (-128) and all zeros in the mantissa. As is shown later, this avoids a problem which occurs when adding a number to zero.

---

<sup>1</sup>The IEEE Standard for Floating-Point Arithmetic uses "denormalized" numbers. A denormalized number consists in an unnormalized mantissa and a special exponent (usually the most negative). It allows the representation of numbers that cannot be represented as normalized numbers due to the fact that the exponent required is smaller than the smallest available exponent. The RAP does not allow denormalized numbers.

| Operand A    | Operand B    | A+B          | A-B            | A×B         |
|--------------|--------------|--------------|----------------|-------------|
| Overflowed   | Overflowed   | Overflowed   | Overflowed     | Overflowed  |
| Overflowed   | Underflowed  | Overflowed   | Overflowed     | Overflowed  |
| Underflowed  | Overflowed   | Overflowed   | Overflowed     | Overflowed  |
| Underflowed  | Underflowed  | Underflowed  | Underflowed    | Underflowed |
| Overflowed   | Normalized # | Overflowed   | Overflowed     | Overflowed  |
| Normalized # | Overflowed   | Overflowed   | Overflowed     | Overflowed  |
| Underflowed  | Normalized # | Normalized # | - Normalized # | Underflowed |
| Normalized # | Underflowed  | Normalized # | Normalized #   | Underflowed |

Table 4.1: Overflow/Underflow Conventions for the Add, Subtract, and Multiply Operations

## 4.2 Design of a 4-bit Adder

The critical component of the circuit design and the one most likely to limit the speed at which the floating-point units can be run is the 4-bit adders used in the design. This is because in the design, signals must propagate through the four bit carry delay and through one or two additional logic levels within one half clock period. A number of 4-bit adders were considered including:

1. A simple precharged Manchester carry chain [36].
2. A precharged Manchester carry chain with positive feedback pulldown circuitry.
3. 4-bit lookahead adder using domino logic [36].
4. Ripple carry adder with optimized carry path.
5. A quaternary full adder [5].

These adders were evaluated in terms of the area they required and the speed they achieved. The quaternary full adder was discarded as a possibility because of area constraints: since it requires encoding the incoming four bit value into a new eight bit value, it doubles the latches and wires in each stage of the datapath. It also requires encoding and

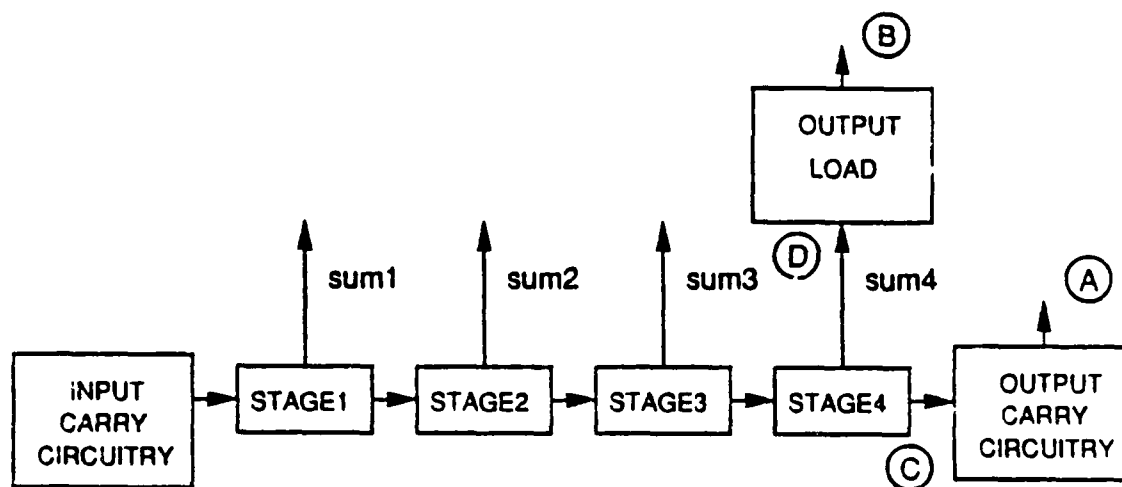


Figure 4.2: General 4-bit Adder SPICE circuit

decoding circuitry which is not convenient when all that is needed is a single 4-bit adder. The remaining four adders were simulated in SPICE. The general form of the circuit used for simulation is shown in Figure 4.2. The parameters varied in order to increase speed were the size of the transistors in the input carry circuitry and the size of the transistors in the carry circuitry itself. Detailed SPICE models, sample waveforms, schematics, and layout for the different 4-bit adders are found in Appendix A.

The signals that are critical in terms of timing are the last carry out which must be latched, and the 4th bit of the sum which is the last bit of the sum to be calculated. Table 4.2 shows the time required for the signals to propagate to points A and B of Figure 4.2, where point A is the point at which the last carry out is stored, and point B is the output of a register in which the most significant bit of the sum is stored. These circuits approximate the worst case capacitive load found in the floating-point circuit design. Because the Ripple Carry adder does not use precharging, its operation is different from the other adders, and the delay is measured to points C and D of Figure 4.2 instead of to points A and B. The speed target for the design requires that the complete calculation happen in one half cycle

| Adder Type                         | Carry | Sum   | Area Estimate    |
|------------------------------------|-------|-------|------------------|
| Manchester carry chain             | 4.2ns | 4.2ns | $48.8K\lambda^2$ |
| Manchester carry chain with kicker | 3.6ns | 3.8ns | $52.4K\lambda^2$ |
| Carry lookahead                    | 2.8ns | 3.2ns | $81.9K\lambda^2$ |
| Ripple Carry                       | 4.4ns | 5.0ns | $47.2K\lambda^2$ |

Table 4.2: 4-bit Adder Delay and Area using CMOS  $2\mu\text{m}$  Worst Speed SPICE Models.

or about 6ns. All four circuits meet this requirement. Also shown in Table 4.2 are area estimates for each 4-bit adder. In order to meet speed goals, the adders use many large transistors (as wide as  $48\mu\text{m}$ ). As a result the area estimates are approximately twice as large as they would be if speed was not crucial, and smaller transistors were used.

The precharged Manchester Carry circuitry and the Manchester carry circuitry with positive feedback pulldown circuitry are shown in Figure 4.3. For a carry of only four bits the precharged Manchester carry chain with positive feedback is marginally better than the simple precharged Manchester carry circuit. Its real benefit is only manifested in carry chains longer than four bits. This circuit also has the disadvantage of having a noise margin equal to the threshold voltage of the p-type transistor: a drop of one threshold on the precharged carry node will activate the positive feedback and pulldown the carry line. The precharged carry lookahead circuit (see Appendix A) is the fastest circuit. The cost is a substantial increase in the area required to implement the circuit because each stage requires its own special carry circuitry which takes inputs from all previous stages. Finally, the ripple carry circuit of Figure 4.4 consists of two different stages, each with transistors sized to minimize delay. It is the slowest in terms of speed, though it requires less area than the Manchester Carry circuit due to the absence of clock lines.

The best choice for the 4-bit adder in terms of satisfying the speed requirements while using the minimum area is the simple Manchester carry chain. This circuit can be optimized

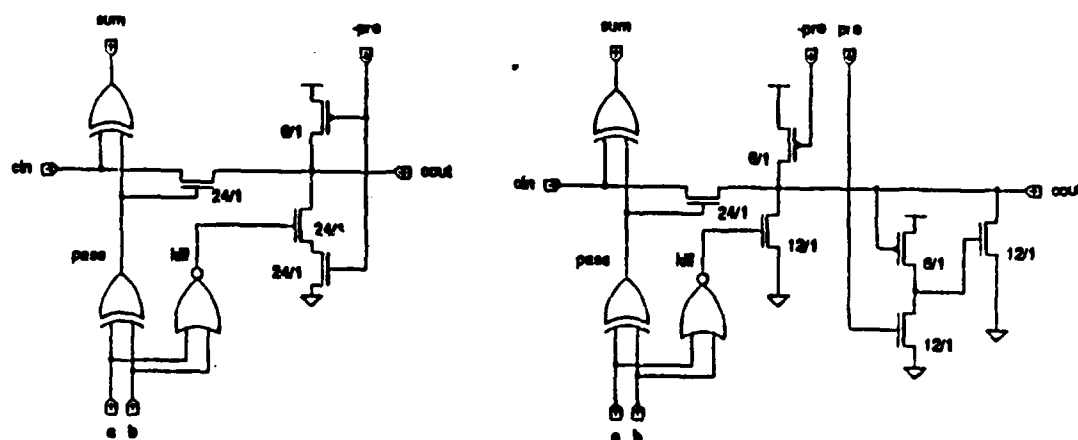


**Figure 4.3: Manchester Carry Circuits a) Normal Manchester Carry Circuit b) Manchester Carry Circuit with Positive Feedback Pulldown Circuit**

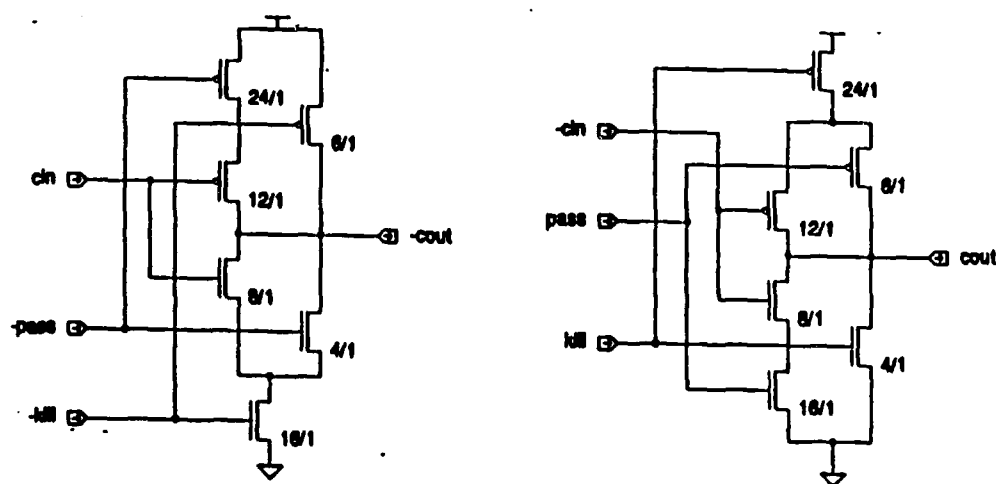


**Figure 4.4: Stages of the Ripple Carry Circuitry**

further by gradually reducing the size of the pulldown transistors at each stage. This works because stages at the end of the carry chain have less capacitance to pulldown than the stages at the beginning of the carry chain, so they can use smaller transistors to achieve the same speed. The gradual reduction in size of the pulldown transistors increases the pulldown speed of the stages at the beginning of the carry chain by reducing the diffusion capacitance present on the carry line.

### 4.3 Floating-Point Units

A serial floating-point adder/subtractor and a serial floating-point multiplier were designed. Both units have a latency of 51 clock cycles of 12.5ns. A new problem can be shifted in as the previous result is being shifted out, so that a complete result is produced once every 51 clock cycles. This corresponds to a rate of 1.57MFlops per unit.

The implementation is 4-bit serial in order to make full use of the clock period. In single bit implementations signals propagate in times much smaller than the smallest clock period that can be reliably distributed without clock skew problems, and thus do not make full use of the clock period. Manipulating four bits at a time means that there is more work to be done at each cycle, and better use is made of the clock period. Efficient serial algorithms exist for doing arithmetic one or two bits at a time [5, 23] and these algorithms have straightforward extensions to doing four bits at a time.

The target clock rate of 80Mhz was chosen for several reasons:

1. Distributing an 80MHz clock over a large CMOS chip is an ambitious but feasible problem to solve.
2. SPICE simulations of the different 4-bit adders indicate that carefully designed 4-bit adders can achieve this target speed.

3. 80MHz is a convenient multiple of 20MHz, the clock rate at which it is expected the RAP memory and control will run.

Doing floating-point serially is more complicated than serial fixed-point because of the of the interaction between the exponent logic and the mantissa logic. In particular, once the exponent and the mantissa of the result have been calculated the number must be normalized: a normalized number in our numbering system is one in which the mantissa  $M$  satisfies  $1/2 \leq |M| \leq 1$ . The mantissa is normalized by shifting it left or right with a corresponding decrement or increment in the exponent value. This interaction requires that results of the exponent logic be stored in latches while the mantissa calculation is taking place. The exponent is then adjusted based on the result of the mantissa calculation. Since the exponent has to wait for the mantissa calculation to be finished before the adjustment can be made, there is increased latency. In the case of a floating-point add, there is additional interaction between the exponent and the mantissa: based on the difference between the exponents of the two numbers, one mantissa must be shifted before it is added to the other mantissa. This interaction between exponent and mantissa calculation is the major cause of complexity in the circuit design.

#### 4.3.1 Floating-Point Adder/Subtractor

The block diagram for the floating-point adder/subtractor is shown in figure 4.5. Operands A and B are fed into the unit four bits at a time, exponent first then mantissa, low order bits first. A start signal initiates the control block which is a large shift register that provides the control signals to various parts of the circuit. The floating-point add or subtract can be divided into the three steps listed below, which are illustrated in Figure 4.6 using a four bit exponent and a twelve bit mantissa.

1. ALIGN STEP. In order to add two floating-point numbers they must be adjusted to have the same exponent. If the difference between the two operands exponents is

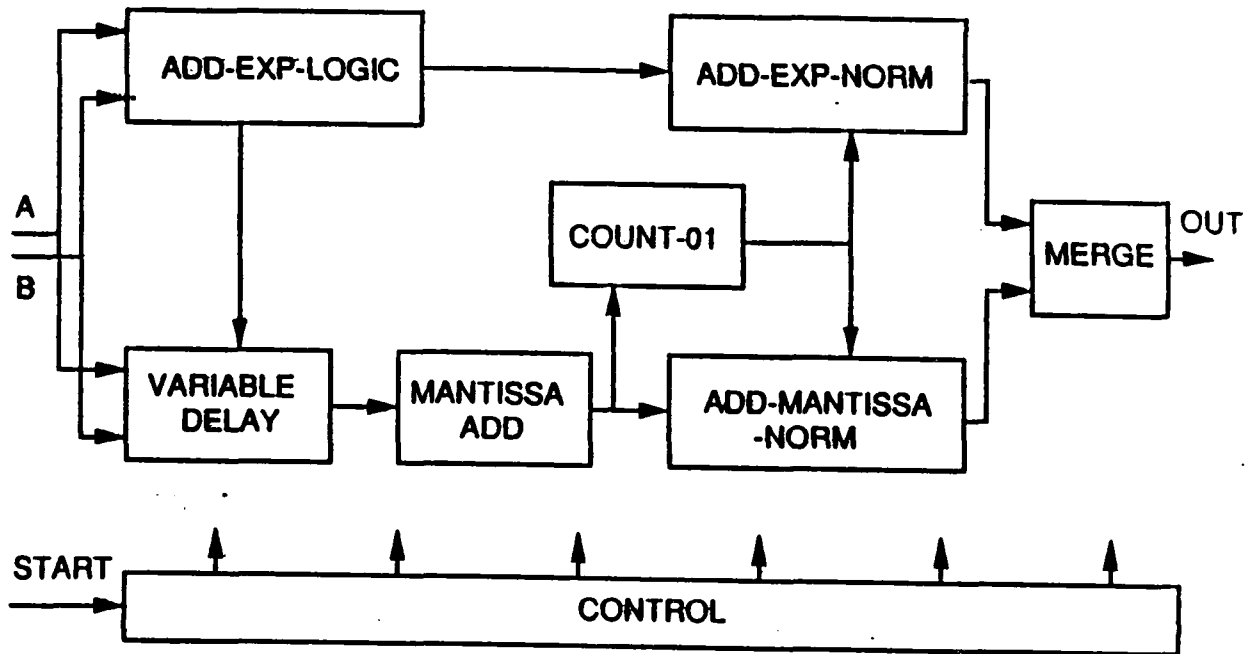


Figure 4.5: Floating-Point Adder/Subtractor Block Diagram

Initial Operands

|    | Exponent | Mantissa                |
|----|----------|-------------------------|
| A: | 0 0 1 0  | 0 1 0 0 0 0 1 1 1 0 0 0 |
| B: | 1 1 1 1  | 1 0 1 1 1 0 1 0 0 1 0 1 |

After Align Step

|    |         |                                                       |
|----|---------|-------------------------------------------------------|
| A: | 0 0 1 0 | 0 1 0 0 0 0 1 1 1 0 0 0                               |
| B: | 0 0 1 0 | <u>1 1 1</u> 1 0 1 1 1 0 1 0 ①<br>sign extend<br>bits |

operand is rounded up

After Add Step

0 0 1 0      0 0 1 1 1 0 1 0 1 1 0 1

After Normalize Step

0 0 0 1      0 1 1 1 0 1 0 1 1 0 1 0

Figure 4.6: Steps in the Floating-Point Add

EXPDIFF the mantissa of the operand with the lowest exponent must be shifted down EXPDIFF bits, dropping the EXPDIFF least significant bits and shifting in EXPDIFF sign extension bits, before it is added to the other mantissa. In the block diagram ADD-EXP-LOGIC subtracts the two exponents and uses the result to control the DELAY block. The DELAY block performs the alignment of the mantissas so that they can be correctly added. In the case of adding zero to a non-zero number, the result should be the non-zero number unchanged. Requiring that the representation of zero have the most negative exponent guarantees that this will occur. If zero was allowed to have an exponent larger than the non-zero number, the ALIGN step would cause the non-zero mantissa to be shifted down and bits would be lost in the final result.

2. ADD STEP. The adjusted mantissas are added or subtracted in the MANTISSA-ADD block, rounding to the nearest number.
3. NORMALIZATION STEP. The result of the mantissa add or subtract may not be normalized and may have to be adjusted. A positive number must have only one leading 0 in the mantissa and a negative number only one leading 1. The COUNT-01 module counts the leading 0s or 1s and provides normalization information to the ADD-EXP-NORM block and the ADD-MANTISSA-NORM block. The ADD-EXP-NORM block takes the largest of the original exponents and subtracts the number of excess mantissa sign bits provided from the COUNT-01 module. The ADD-MANTISSA-NORM removes the excess sign bits from the mantissa result. In the case that the result of the MANTISSA-ADD overflows, the exponent is incremented and the correct sign is added to the mantissa.

Logic is included to take care of overflow and underflow, both in the case where the operands have already overflowed or underflowed, and in the case that the calculation itself causes an overflow or underflow. Detailed schematics of the floating-point adder/subtractor are found in [12].

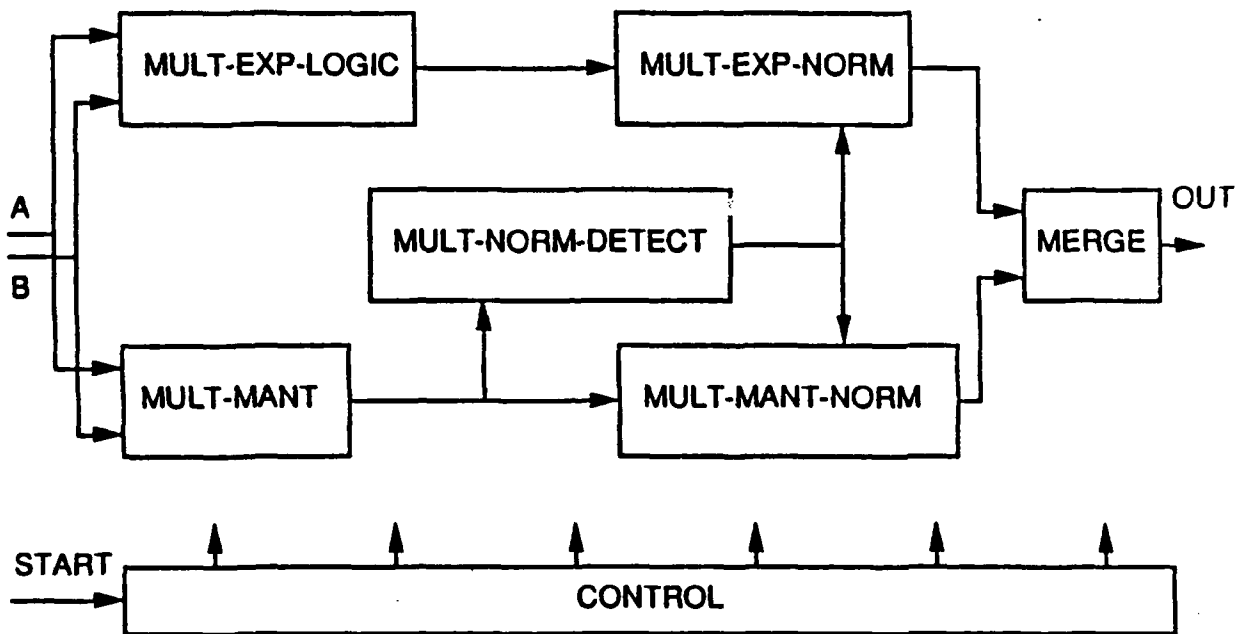


Figure 4.7: Floating-Point Multiplier Block Diagram

#### 4.3.2 Floating-Point Multiplier

The block diagram for the floating-point multiplier is shown in figure 4.7. Operands A and B are fed into the unit four bits at a time, exponent first then mantissa, low order bits first. A start signal initiates the control block in the same way as in the adder/subtractor. The floating-point multiply can be divided into the two steps listed below, which are illustrated in Figure 4.8 using a four bit exponent and a twelve bit mantissa.

1. **CALCULATE STEP.** The exponents are added in the MULT-EXP-LOGIC and mantissas are multiplied in the MANTISSA-MULT. These two operations are independent.
2. **NORMALIZE STEP.** The result of the mantissa multiply can have one or two extra sign bits. One extra sign bit occurs because when multiplying two two's complement mantissas, the full precision result always has an extra sign bit. This sign bit is dropped and an extra low order bit is added to the mantissa. If there are two extra

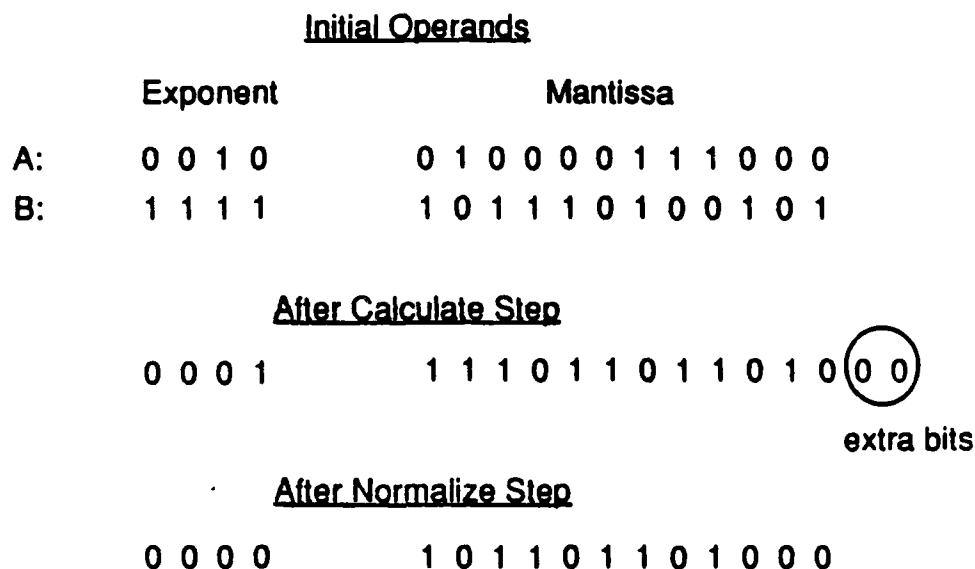


Figure 4.8: Steps in the Floating-Point Multiply

sign bits, then both extra sign bits are dropped and the exponent is decremented.

Logic is included to take care of overflow and underflow, both in the case where the operands have already overflowed or underflowed, and in the case that the calculation itself causes an overflow or underflow. Detailed schematics of the floating-point multiplier are found in [12].

### Mantissa Multiply Algorithm

The algorithm used to multiply the mantissas is an extension of an algorithm described by Lyon in [23]. This algorithm does multiplication of two two's complement numbers using Booth encoding [2, 18]. The extensions and modifications to the algorithm include:

1. Doing four bits of each operand at a time rather than one or two bits at a time.
2. Doing Booth encoding as a separate circuit at the beginning of the multiplier pipeline.

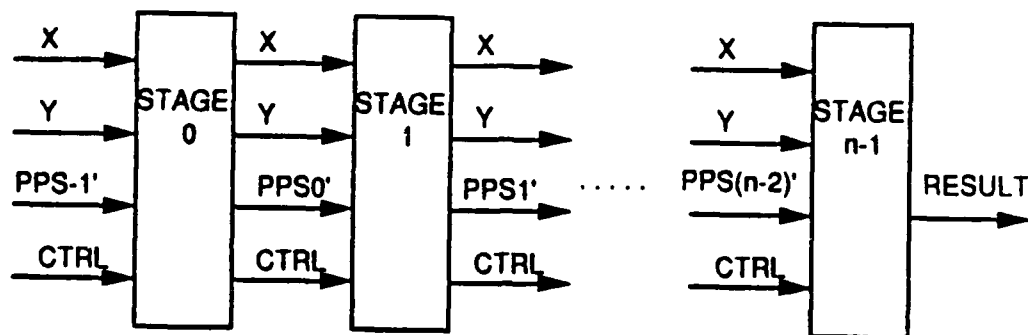


Figure 4.9: Mantissa Multiply Pipeline

3. Design of a special final stage which eliminates the extra sign bit which occurs when multiplying two's complemented numbers (there is always at least one extra sign bit except for the special case when minus one is multiplied by minus one).

Full length multiplication of  $N$  bit operands gives results of  $2N$  bits long. Practically, if the operands are considered as fractions between  $-1$  and  $1$ , this means that the bottom  $N$  bits must be truncated in order to represent the result in an  $N$  bit field. In Lyon's algorithm each partial product and partial product sum is restricted to  $N$  bits. Figure 4.9 shows how the pipeline is set up. Figure 4.10 shows an example of a multiply of two positive binary fractions, that gives a flavor of how the algorithm proceeds. Each stage of the multiplier is responsible for generating and adding a partial product (PP) to the partial product sum (PPS), and is responsible for discarding the bottom bits of this result. In the case that one bit is done at a time, each stage generates a partial product, adds it to the sum of partial products coming from the previous stage, and passes this result to the next stage truncated by one bit. The final stage outputs the  $N$  bit result.

Doing two's complement numbers is more complicated because partial products can be negative, and have implicit sign extension bits that extend infinitely to the left. For a detailed description of two's complement multiply algorithms and the handling of the implicit sign extension, the reader is referred to Lyon's paper [23].

|        |                |
|--------|----------------|
| 0.1011 |                |
| 0.1101 |                |
| <hr/>  |                |
| 0000   | PPS-1          |
| 1011   | PP0            |
| <hr/>  |                |
| 1011   | PPS0           |
| 0101   | PPS0' truncate |
| 0000   | PP1            |
| <hr/>  |                |
| 0101   | PPS1           |
| 0010   | PPS1' truncate |
| 1011   | PP2            |
| <hr/>  |                |
| 1101   | PPS2           |
| 0110   | PPS2' truncate |
| 1011   | PP3            |
| <hr/>  |                |
| 10001  | PPS3           |
| 1000   | PPS3' truncate |
| 0000   | PP4            |
| <hr/>  |                |
| 0.1000 | RESULT         |

Figure 4.10: Example Multiply of Two Positive Fractions

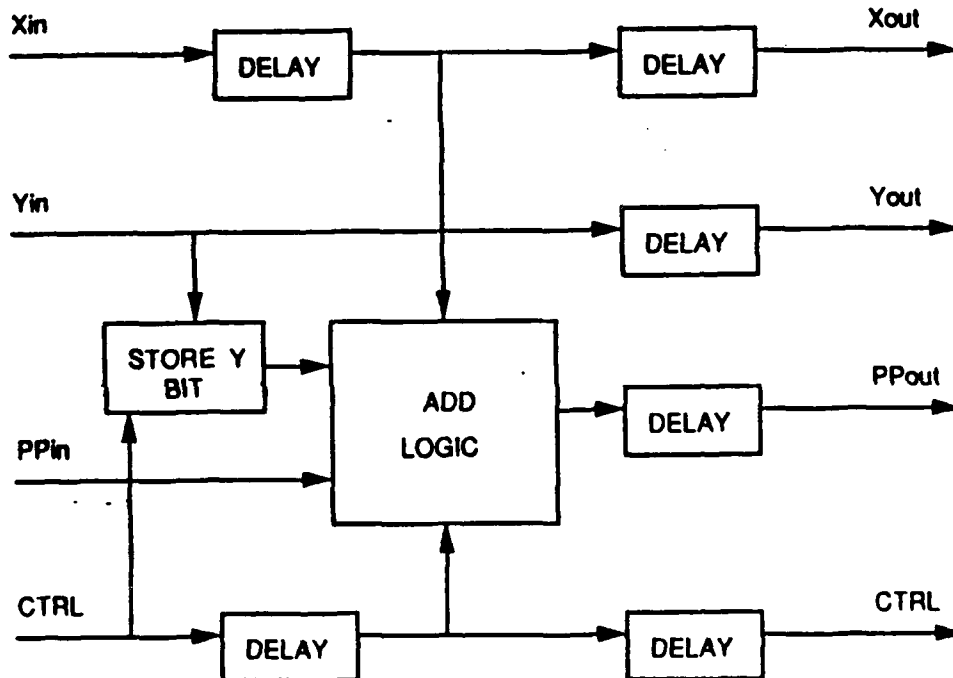


Figure 4.11: Simplified Multiply Cell

Figure 4.11 shows a simplified view of what the multiplier stage looks like. Operands are shifted in least significant bit first. Each "digit"  $Y_i$  (which can consist of one, two, or four bits) of the multiplier gets stored in the  $i$ th pipeline stage and generates the  $i$ th partial product by looking at the digits  $X_i$  of the multiplicand. The  $i$ th partial product then gets added to the incoming partial product sum. Note that to generate each partial product, the digit of the multiplier must "see" all bits of the multiplicand e.g. digit  $Y_1$  of the multiplier must get to pipeline stage 1 before the least significant digit  $X_0$  of the multiplicand, so that the correct partial product can be generated. This is a problem if both operands are being shifted in at the same time with the same delay: any given digit of the multiplier will never catch up to the digits of the multiplicand that were shifted in before it. To solve this problem the path of the multiplicand is made to be twice as slow as the path of the multiplier by inserting an extra delay element. This allows the  $i$ th multiplier digit to reach the  $i$ th stage before the least significant digit of the multiplicand. The control signal is used to control when the  $Y_i$  digit is latched.

If only a single bit is done at a time, then  $N$  stages are required for  $N$  bit operands, one stage for each bit of the multiplier. If instead two bits of the multiplier are shifted in at once, then the logic for these two bits can be combined and the number of stages is reduced to  $N/2$ . Furthermore, doing two bits at once means that modified Booth encoding can be used. This is a technique in which each partial product to be added is generated without requiring another adder, but requires only simple shift and invert operations. The two bit case can be extended to doing four bits. This is done by taking the four bits of the multiplier operand, dividing it into two groups of two bits which are each modified Booth encoded. Then two partial products are calculated, one associated with each of the Booth encoded fields. These two partial products are added to the partial product sum in a single clock cycle, with one partial product being added on the first phase of the clock and the other one on the second phase. The cell is complicated by the fact that the multiplicand arrives in groups of four bits, meaning that six bits of each of the two partial product within a cell are calculated in one clock cycle. Furthermore these partial products overlap by four

| Cell                   | Area             |
|------------------------|------------------|
| Static Register        | $3706\lambda^2$  |
| Shift Register         | $2703\lambda^2$  |
| Gate                   | $1657\lambda^2$  |
| Control Shift Register | $16218\lambda^2$ |
| 4-bit Adder            | $48416\lambda^2$ |

Table 4.3: Basic Cell Area Estimates

bits. The interested reader is referred to the schematics of [12] for details.

#### 4.3.3 Area Estimates

In order to estimate area, the area for a number of basic cells was determined by laying them out or by looking at previous layouts. These cells are listed in Table 4.3. In this Table, the estimate used for the area of a gate is one half the area of an XOR gate, which is larger than the area required by simple one or two input gates, but smaller than most complex gates. This estimate is conservative since most of the gates in the circuit are inverters, pass transistors, 2 input NOR gates, and 2 input NAND gates. The Control Shift Register is a special register used by the control circuitry, and is estimated to require the area of 6 shift registers.

For each of the floating-point units, the number of each type of basic cell was counted and used to give an area estimate for the entire unit. The area of sub-circuits which did not fit into one of the categories was approximated by looking at how many of the basic cells would take up the same area. Table 4.4 shows the break up of area for each of the floating-point units. 20% additional area is included for wiring.

| Cell                   | Adder/Subtractor |                       | Multiplier      |                       |
|------------------------|------------------|-----------------------|-----------------|-----------------------|
|                        | Number           | Area ( $K\lambda^2$ ) | Number          | Area ( $K\lambda^2$ ) |
| Static Register        | 90               | 334                   | 33              | 122                   |
| Shift Register         | 291              | 787                   | 598             | 1616                  |
| Gate                   | 255              | 423                   | 370             | 613                   |
| Control Shift Register | 43               | 697                   | 51              | 827                   |
| 4-bit Adder            | 8                | 387                   | 30              | 1452                  |
| 20% Wiring             | X                | 526                   | X               | 926                   |
| Total Area             | $3.2M\lambda^2$  |                       | $5.6M\lambda^2$ |                       |

Table 4.4: Floating-Point Unit Area Estimate

#### 4.4 Other Hardware Components

The remaining hardware components of the datapath are the registers and the switch. Input and output register cells which perform parallel to serial and serial to parallel conversion respectively are shown in Figure 4.12.

The switch circuitry is shown in Figure 4.13. Precharged lines going to the arithmetic units are conditionally connected to one of the inputs based on the decoder lines.

#### 4.5 Hardware Improvements

A major disadvantage of the current floating-point unit design is that two problems cannot be pipelined one immediately after the other. The units have a pipeline latency of 51 clock cycles. This is just over three word times, where a word time is 16 clock cycles and corresponds to the time required to shift a complete operand into an AU. Ideally it should be possible to look at the unit as a three stage pipeline, and have three problems in the unit at once. Currently there must be a two word time gap between when consecutive problems get fed into the units, leading to a performance which is only a third of that which is possible.

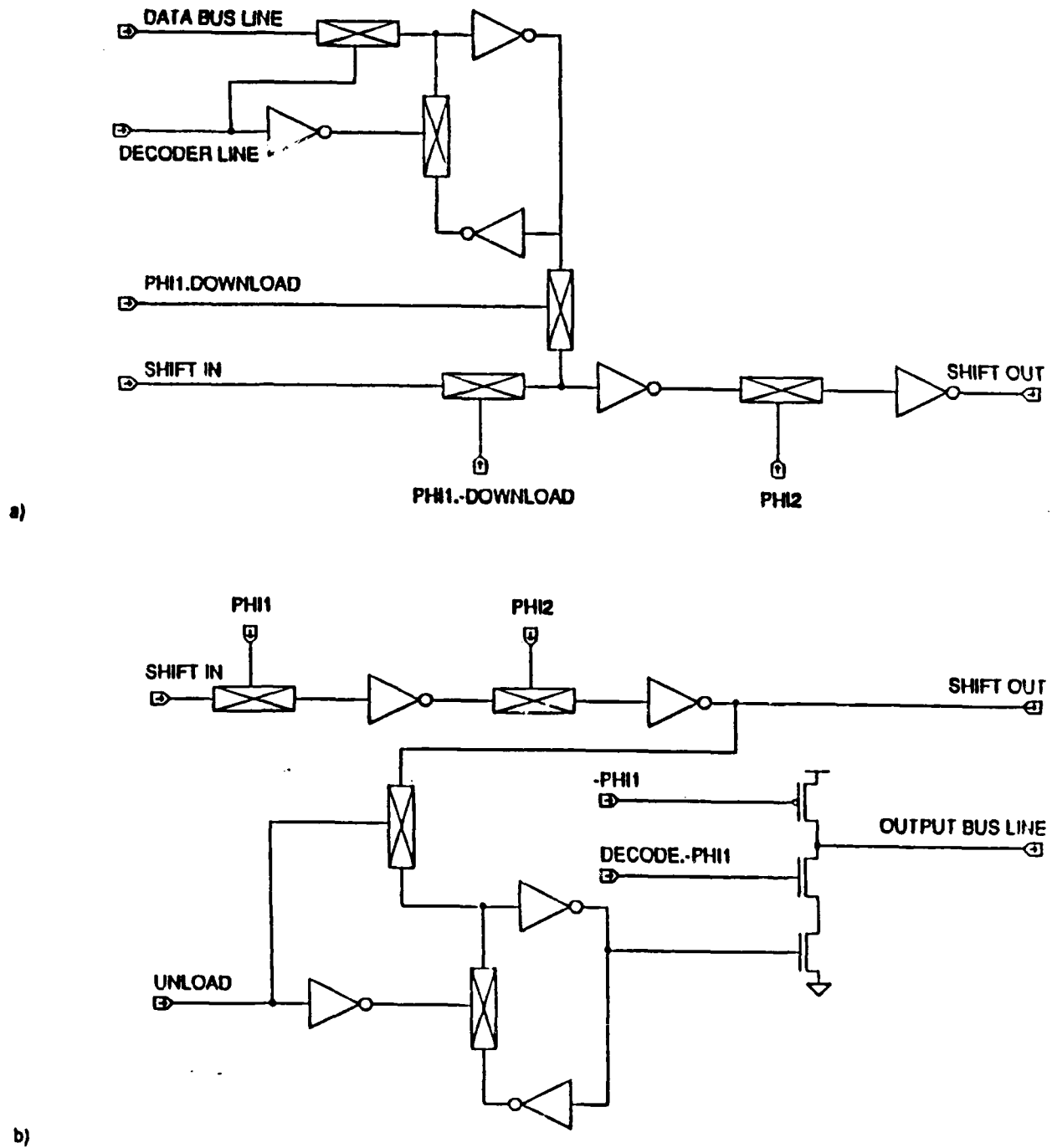


Figure 4.12: a) Input Register Cell b) Output Register Cell

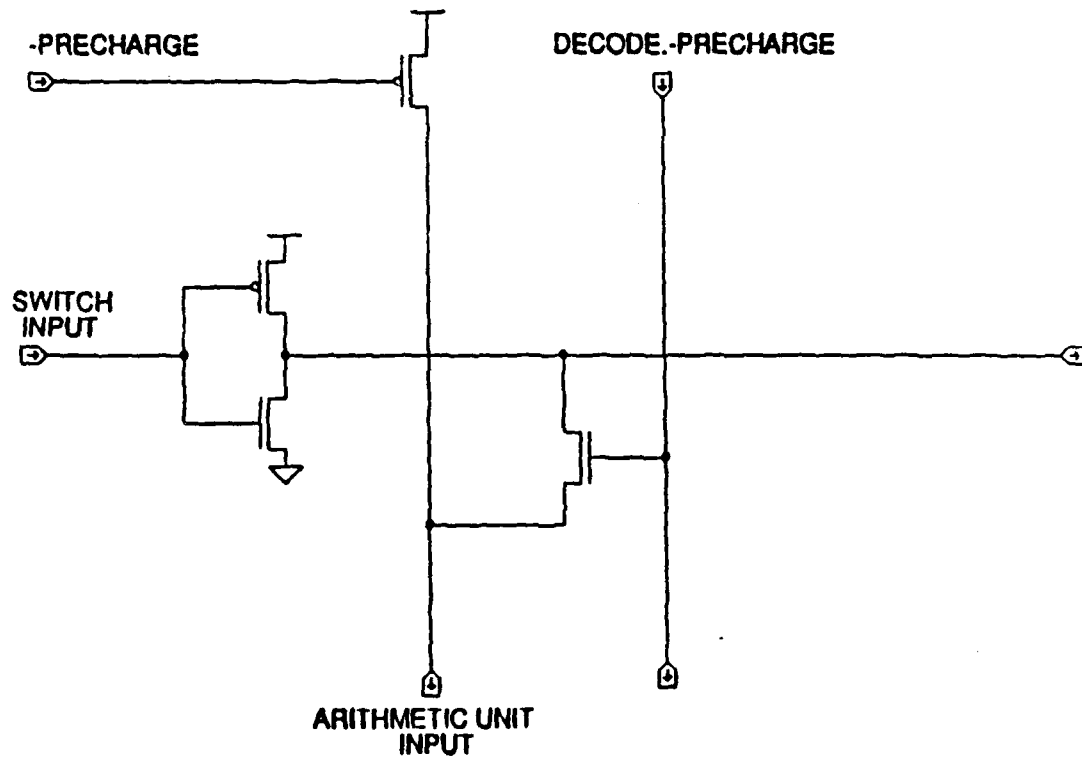


Figure 4.13: Switch Cross Point

Fortunately, the units can easily be modified to allow this type of pipelining (these modifications were not implemented due to time constraints). The problem which currently prevents this type of pipelining is that certain values are stored in register cells (e.g. the exponent waiting for the normalization step), and the contents of these registers must not be written over by values coming from the next problem. This problem is solved by having two levels of storage, and in some cases three levels, so that no information is lost.

In the case of the adder/subtractor, a new variable delay module needs to be designed. The variable delay is currently designed as a shift register with a variable number of stages: if the first problem entering is delayed more than the following problem then they will interfere with each other. The simplest solution to this is to design the variable delay as a shift register with constant delay, but where the output can be tapped from different points, thus giving a variable delay. The sign extension logic is moved to the output of this shift register so that when the most significant bit gets shifted through, all bits thereafter will be sign extension bits.

## 4.6 RAP Fixed-Point Prototype

A RAP test chip (Figure 4.14) that does 16 bit fixed-point arithmetic has been designed by MIT students Stuart Fiske, Josef Shaoul, and Petr Spacek in order to investigate some of the ideas described above, in particular the idea of having serial arithmetic AUs in a reconfigurable datapath. The chip was fabricated and tested in MOSIS  $3\mu\text{m}$  Scalable CMOS technology.

The block diagram of the fixed-point RAP is shown in Figure 4.15. It consists of a bank of eight 16 bit input registers, twelve 9 bit switch configuration registers, a datapath consisting of a switch and several AUs, and eight 16 bit output registers. The datapath is a three stage pipeline, each stage uses four AUs and is connected to the next stage by a statically reconfigurable sparse crossbar switch, as shown in Figure 4.16. The AUs are

16 bit, two-bit serial Arithmetic Units. Each AU takes three operands and is capable of doing multiplication, addition, subtraction of two of its operands while passing the third unchanged, or of multiplying two of its operands and adding/subtracting the third.

Operation proceeds as follows: the switch registers are loaded using the bit parallel input bus and then feed statically into the switch to determine the switch configuration and the AU functionality. The input registers are also loaded using the input bus, and then shift serially into the first stage of the switch. Calculations take place as the operands and intermediate results propagate through the three stages of the datapath, until the results are shifted into the output registers. Finally, results are unloaded onto the bit parallel output bus. New input operands can be loaded into the input registers as the operands from the previous problem are shifting into the datapath, and similarly, the results from the previous problem can be unloaded from the output registers as results are shifting out of the datapath. This means problems can be pipelined and a new problem can begin once every nine clock cycles.

Although the switch setup is different than that of the floating-point RAP, this chip demonstrates that the switch can be efficiently implemented: about 12% of the total chip area is devoted to the switch and switch control. This percentage will be much smaller in the case of the 64 bit floating-point operations because the AUs and registers will be much bigger.

The chip was tested up to 8.33MHz giving a peak performance of 0.93Mops (Mega operations per second) per functional unit, and 11.1Mops for the entire chip. Testing speed was restricted by the limits of the test apparatus used.

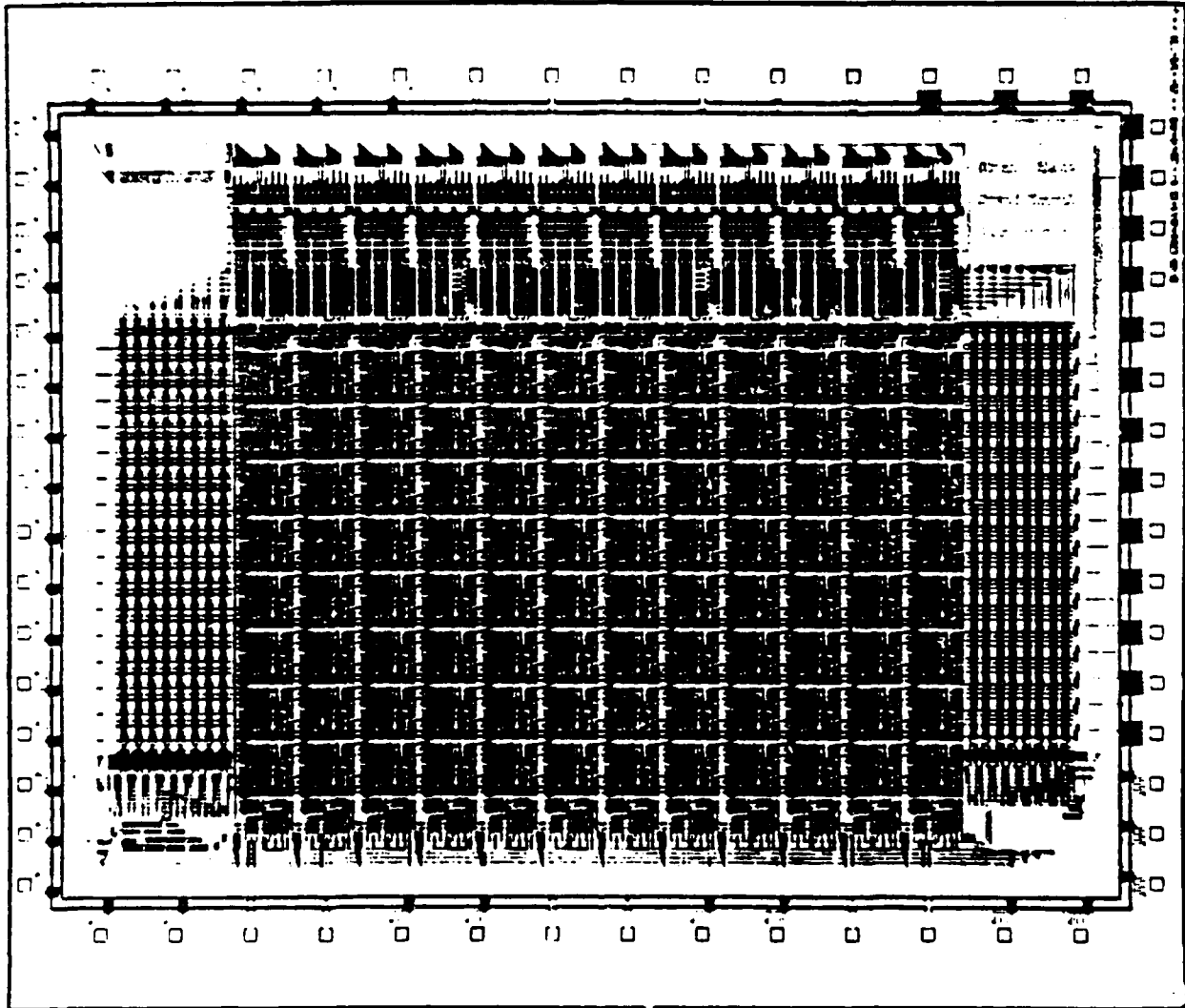


Figure 4.14: RAP Fixed-Point Prototype

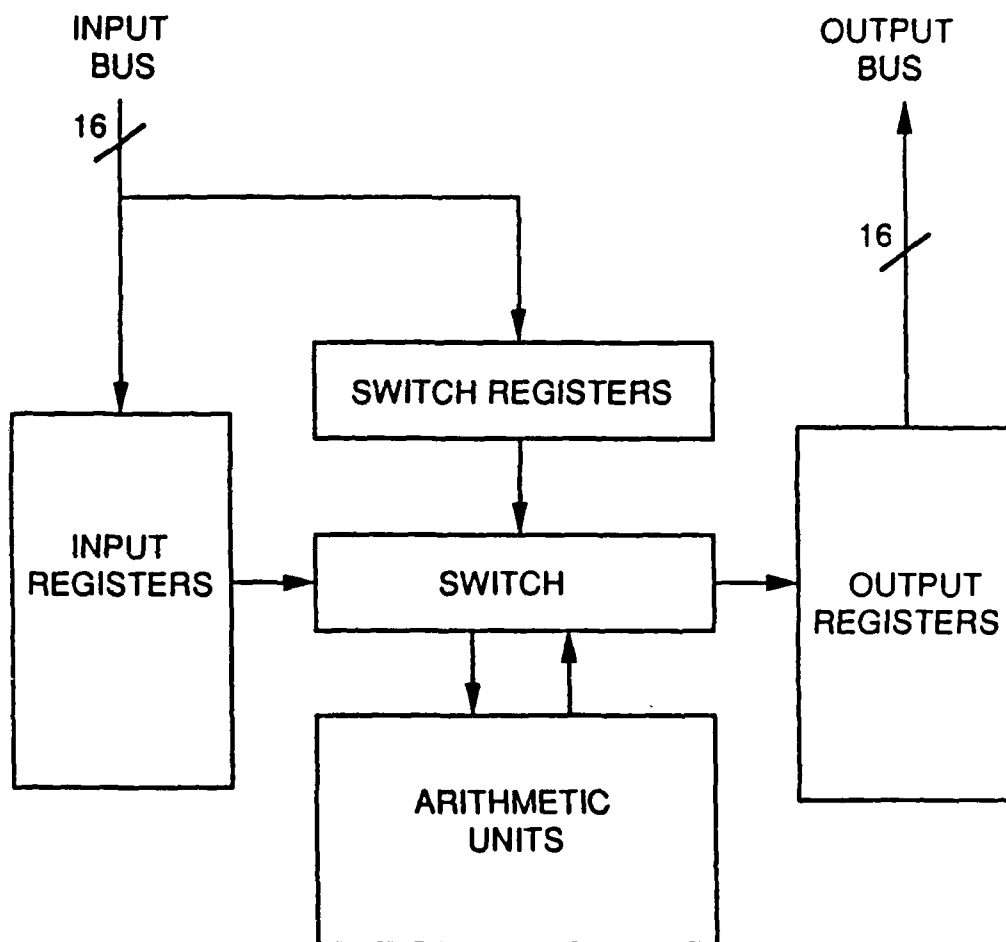
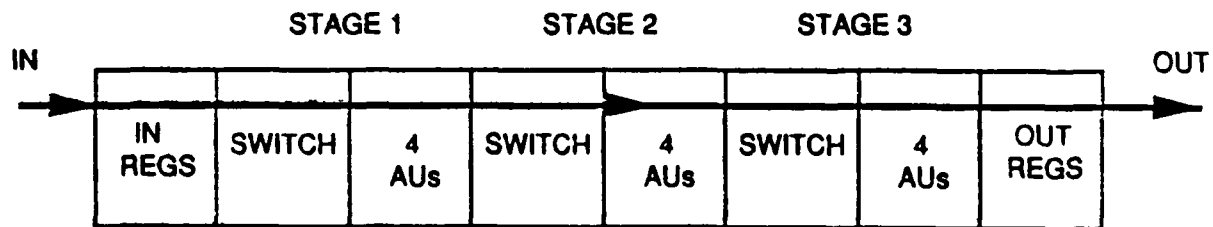
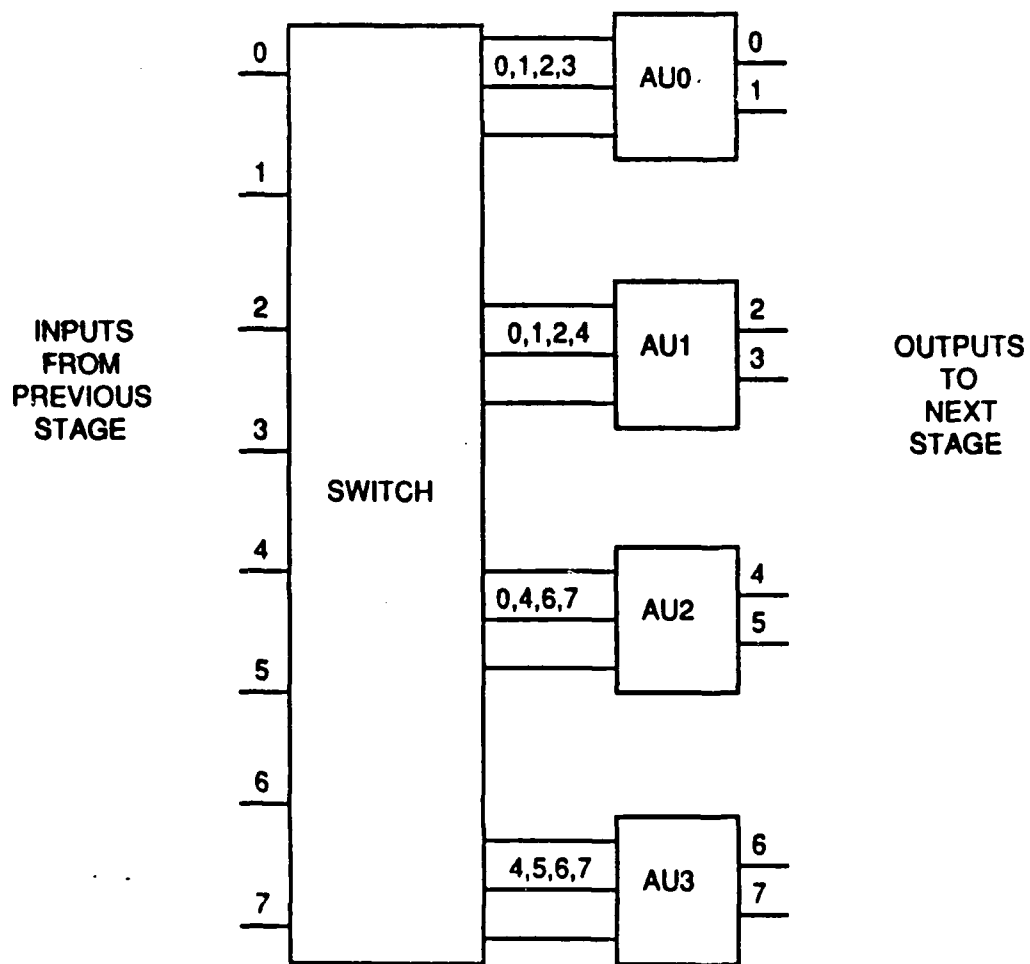


Figure 4.15: Block Diagram of the Fixed-Point RAP



a)



b)

Figure 4.16: Fixed Point-RAP Datapath a) Three Stage Pipeline, b) Switch Connectivity at Each Stage

## 4.7 Summary

This Chapter addressed some of the more important issues of the circuit design of the RAP datapath. The schematic level design of a 4-bit serial floating-point adder/subtractor and a 4-bit serial floating-point multiplier has been carried out. The floating-point number representation used in the design is developed, based on a format consisting in an 8 bit two's complement exponent field, and a 56 bit, two's complement normalized mantissa field. Methods for dealing with special conditions such as overflow and underflow are described. Based on a study of different  $2\mu\text{m}$  CMOS 4-bit adders using SPICE, it is expected that the floating-point units will run at 80MHz. The current design of the floating-point units has a performance of 1.57MFlops, while a pipelined design would increase this performance to 4.70MFlops. The area estimate for the floating-point adder/subtractor is  $3.2M\lambda^2$ , and for the multiplier  $5.6M\lambda^2$ .

The fixed-point RAP is a chip that was designed to do 16 bit fixed-point arithmetic using a statically reconfigurable datapath. This chip demonstrates some of RAP ideas using fixed-point arithmetic.

## Chapter 5

# Expression Compiler

*The future enters into us, in order to transform  
itself in us, long before it happens.*

— RAINER MARIA RILKE, in *Letters to a Young Poet*

*Bless thee, Bottom! bless thee! thou art translated.*

— SHAKESPEARE in *A Midsummer-Night's Dream*, III, i, 124

This Chapter describes a compiler that maps an arithmetic expression into a series of switch configurations (a method) that are used by the RAP to calculate the expression. The compiler serves several purposes: First, it generates methods for the benchmark expressions used in Chapter 6 to evaluate the performance of the RAP. The number of switch configurations required to evaluate each expression determines the performance of the chip. Second, the compiler allows the performance comparison of different resource configurations: number of functional units available and switch connectivity. Third, the compiler allows the investigation of different ways in which the compilation process itself can help improve performance.

Section 5.1 discusses the scheduling problem that the compilation process addresses. Section 5.2 describes the search algorithm used, including pruning techniques. Finally, possible enhancements to the compiler and other possible approaches to the problem are discussed in section 5.3. The compiler is written in Common LISP and the code is found in [11].

## 5.1 The Problem

The problem that must be solved is the following: given an expression that contains a number of add, subtract, and multiply operations, find how these operations can be scheduled on the functional units in the RAP datapath. Scheduling must take into account the limited number of functional units and the limited connectivity between the functional units and the inputs. The problem reduces to a graph matching problem in which the Directed Acyclic Graph (DAG) [1], representing the expression, is mapped onto the resource graph, that represents the functional units and their connectivity. In defining the resource graph it is useful to first define a "level" a level is a snapshot of the datapath state (i.e. which operations are being computed in which functional units) for a given word time. Each level has inputs coming from the "previous level" and has outputs that are feeding into the "next level". The resource graph is a multiple level graph used to represent the use of the functional units over time. The inputs of a each level are the inputs of the functional units, the outputs of each level are the outputs of the functional units, and the inputs of each level are connected to the outputs of the previous level as described by the switch connectivity.

An optimal schedule is one that requires the smallest amount of time to complete the calculation. Optimal scheduling with finite resources is a well known NP-complete problem [13]. However, simple heuristic scheduling methods such as list scheduling have been found to generate near optimal solutions [10]. The basic idea in list scheduling is to assign priorities to the nodes in the DAG, based for example on their maximum distance from a DAG output.

Scheduling then proceeds by scheduling as many operations as possible starting with the highest priority operations, until resource constraints prevent further scheduling. All nodes are scheduled in the same way until all tasks have been completed.

The scheduling problem is harder in the case of the RAP because of the limited connectivity of the switch. A choice of schedule at one level may make it impossible to complete the calculation of the expression. The limited connectivity eliminates the use of equivalence classes [1] as a means of simplifying the scheduling task since any two units that have equivalent functionality (there are four adders/subtractors for instance) do not have equivalent connectivity. Often, at a given level, two values that must be added or multiplied are not both connected to a common adder or multiplier. This means that the computation must be delayed and the operands must be fed through feedthroughs. Furthermore, the feedthroughs used should be chosen so that on the next cycle it will be possible to schedule the operation. Now if a different set of schedules had been chosen for the previous levels, it is possible that the extra feedthrough operations would not have been required. This example illustrates that it is important not only to determine which operations should be scheduled at a given level, but also which specific units should be used. Fortunately, the limited connectivity does provide an inherent limitation on the search tree, since many schedules are impossible.

## 5.2 The Algorithm

The expression compiler does a tree search for possible mappings of an expression onto the switch. First the expression is transformed into a DAG in which all common subexpressions have been combined. Then, a depth first search is used that assigns operations to AUs and intermediate results to feedthroughs until all the final results are available at the outputs of functional units. This assignment of operations to AUs and intermediate results to feedthroughs at a given level is referred to as a schedule for that level. If at any point the search gets stuck and can no longer advance (for instance if there are more intermediate

results that have to be fed through than there are feedthrough units) the search backs up to the previous level and tries a new schedule. This continues until a method which solves the problem is found. If the method found is not optimal, then the search continues to try and improve on the method already found. A complete search is very expensive because the problem is exponential and various techniques are used to limit the search as is discussed in Section 5.2.3. Since the search is limited by heuristics, an optimal solution is not guaranteed.

### 5.2.1 An Example

It is useful to look at an example. Suppose the compiler must schedule the three expressions  $(*(+ 3 X) (+ Y X))$ ,  $(* (+ Z Y) (+ Z H))$ ,  $(*(+ H 2) 4)$ . The compiler first transforms this list of expressions into a DAG as shown in Figure 5.1. Each node is given a number, and has pointers to its inputs and to its destination nodes. The compiler then begins scheduling the operations in time and space: it determines for each word time which operations are to be scheduled on which functional units.

Figures 5.2 and 5.3 represent output from the compiler. For each word time the level information is illustrated, which consists of two parts: the schedules of operations that begin in that word time, and an array showing the current position of all active operations in the functional units. The three schedules for each word time are the add/subtract schedule, the multiplier schedule, and the feedthrough schedule. Each entry in the schedule is a pair of numbers consisting of the node number that is being scheduled (taken from the DAG), and the number of the functional unit of that type it is being scheduled on. The functional units of a given type are assigned numbers 0 through  $(n - 1)$  where  $n$  is the number of functional units of that type. If an operation has both of its operands ready but cannot be scheduled, it is assigned to the number  $n$ . For instance, in word time 1, nodes 12, 9, 7, and 4 are scheduled on add/subtract units 0 through 3. Operation 2 is assigned to add/subtract unit 4 which indicates that it has not been scheduled due to the insufficient number of adders.

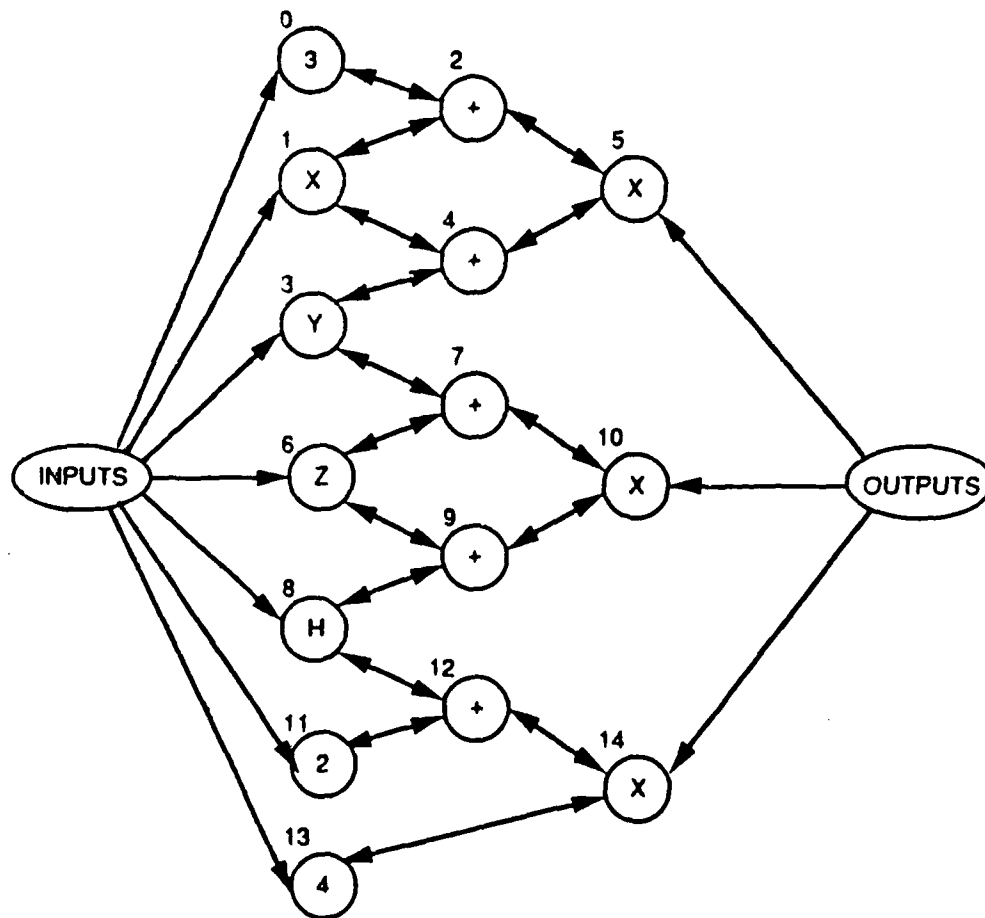


Figure 5.1: DAG for the List of Expressions  $(+ 3 X)$ ,  $(+ Y X)$ ,  $(+ Z Y)$ ,  $(+ Z H)$ ,  $(+ H 2)$ ,  $(+ 4)$ .

Word Time 1:

Adder/Subtractor schedule: (12 0) (9 1) (7 2) (4 3) (2 4)

Multiplier schedule:

Feedthrough schedule: (13 0) (1 3) (0 1)

```

0 12 NIL NIL
1 13
2 NIL NIL NIL
3 0
4 9 NIL NIL
5 NIL
6 NIL NIL NIL
7 1
8 7 NIL NIL
9 NIL
10 NIL NIL NIL
11 NIL
12 4 NIL NIL
13 NIL
14 NIL NIL NIL
15 NIL

```

Word Time 2:

Adder/Subtractor schedule: (2 0)

Multiplier schedule:

Feedthrough schedule: (13 0)

```

0 2 12 NIL
1 13
2 NIL NIL NIL
3 NIL
4 NIL 9 NIL
5 NIL
6 NIL NIL NIL
7 NIL
8 NIL 7 NIL
9 NIL
10 NIL NIL NIL
11 NIL
12 NIL 4 NIL
13 NIL
14 NIL NIL NIL
15 NIL

```

Word Time 3:

Adder/Subtractor schedule:

Multiplier schedule:

Feedthrough schedule: (13 0)

```

0 NIL 2 12
1 13
2 NIL NIL NIL
3 NIL
4 NIL NIL 9
5 NIL
6 NIL NIL NIL
7 NIL
8 NIL NIL 7
9 NIL
10 NIL NIL NIL
11 NIL
12 NIL NIL 4
13 NIL
14 NIL NIL NIL
15 NIL

```

Figure 5.2: DAG Schedule (part 1)

Word Time 4:

Adder/Subtractor schedule:

Multiplier schedule: (14 0) (10 1)

Feedthrough schedule: (4 3)

0 NIL NIL 2  
 1 NIL  
 2 14 NIL NIL  
 3 NIL  
 4 NIL NIL NIL  
 5 NIL  
 6 10 NIL NIL  
 7 4  
 8 NIL NIL NIL  
 9 NIL  
 10 NIL NIL NIL  
 11 NIL  
 12 NIL NIL NIL  
 13 NIL  
 14 NIL NIL NIL  
 15 NIL

Word Time 5:

Adder/Subtractor schedule:

Multiplier schedule: (5 0)

Feedthrough schedule:

0 NIL NIL NIL  
 1 NIL  
 2 5 14 NIL  
 3 NIL  
 4 NIL NIL NIL  
 5 NIL  
 6 NIL 10 NIL  
 7 NIL  
 8 NIL NIL NIL  
 9 NIL  
 10 NIL NIL NIL  
 11 NIL  
 12 NIL NIL NIL  
 13 NIL  
 14 NIL NIL NIL  
 15 NIL

Word Time 6:

Adder/Subtractor schedule:

Multiplier schedule:

Feedthrough schedule:

0 NIL NIL NIL  
 1 NIL  
 2 NIL 5 14  
 3 NIL  
 4 NIL NIL NIL  
 5 NIL  
 6 NIL NIL 10  
 7 NIL  
 8 NIL NIL NIL  
 9 NIL  
 10 NIL NIL NIL  
 11 NIL  
 12 NIL NIL NIL  
 13 NIL  
 14 NIL NIL NIL  
 15 NIL

Word Time 7:

Adder/Subtractor schedule:

Multiplier schedule:

Feedthrough schedule: (10 1) (14 0)

0 NIL NIL NIL  
 1 14  
 2 NIL NIL 5  
 3 10  
 4 NIL NIL NIL  
 5 NIL  
 6 NIL NIL NIL  
 7 NIL  
 8 NIL NIL NIL  
 9 NIL  
 10 NIL NIL NIL  
 11 NIL  
 12 NIL NIL NIL  
 13 NIL  
 14 NIL NIL NIL  
 15 NIL

Figure 5.2: DAG Schedule (part 2)

The second part of the diagram, represents the progress of the operations through the functional units. Each of the sixteen entries in the table represents a functional unit (AU or feedthrough) and its state. The state of a functional unit is described by which nodes in the DAG it is working on. In the case of an adder/subtractor or a multiplier, three operations can be in progress at once, because of the three word time latency of the units. A feedthrough only has a latency of one word time. In the diagrams, operations are designated by their DAG number, and a NIL entry indicates that no operation is in progress. Each of the functional units of a given type has a position in the overall bank of functional units. For instance, the adders/subtractors 0 through 3 are in positions 0,4,8, and 12 in the bank of functional units. In the first word time of the method, nodes 12, 9, 7 and 4 are in the first word delay of the four add/subtract units. Input operands 0, 1 and 13 are fed through feedthrough units. In the second word time of the method, nodes 12, 9, 7, and 4 have "advanced" one word time, node 2 is scheduled in an add/subtract unit, and node 13 is again fed through. At the end of word time 3, results begin to come out of the adders/subtractors which allows the scheduling of multiply operations in word time 4. Calculation continues until all results are available at outputs of functional units in word time 7.

The switch configuration for each level is determined by the schedules for each word time. The compiler also outputs the schedule as a method that can be used by the RAP simulator. Information is included as to which input registers are to be loaded with operands, and which output registers will contain the results.

### 5.2.2 Sequencing Schedules

A way must be found to sequence through schedules at a given level. When first scheduling a level, the compiler determines all the possible adds, subtracts, and multiplies that can be done based on the outputs available from the previous level. When scheduling or rescheduling a level, the add/subtract operations are assigned to add/subtract units (this

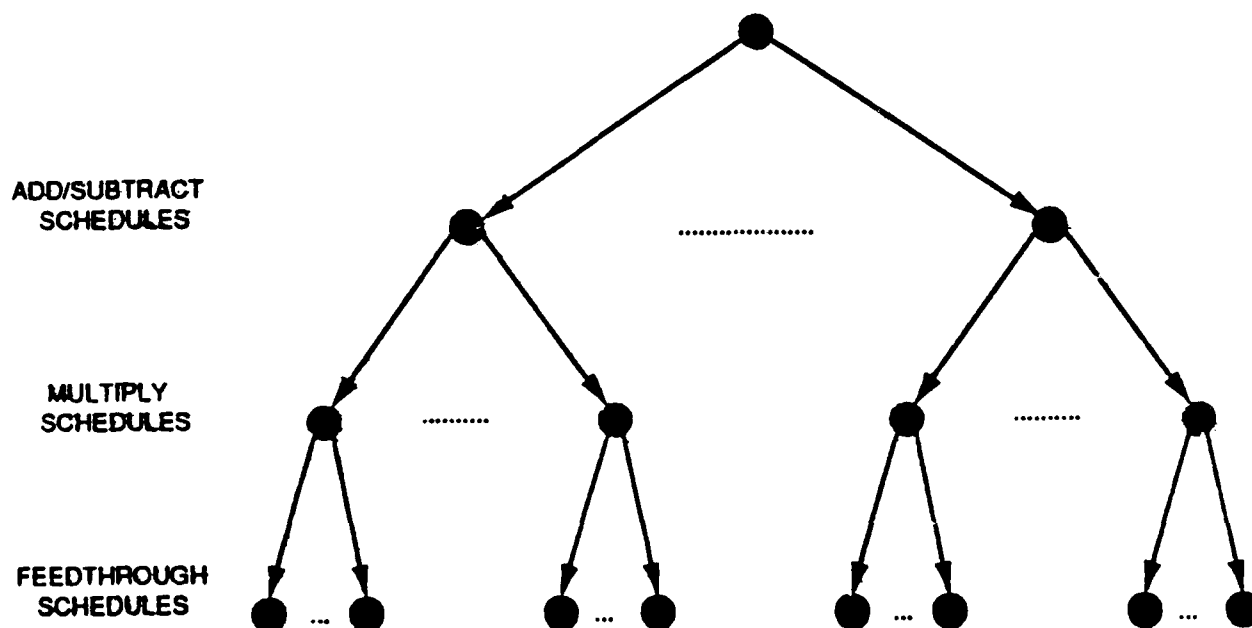


Figure 5.4: Tree of All Possible Schedules for a Level

is called the add/subtract schedule), and the multiplies are assigned to multiply units (the multiply schedule). Once the operations have been scheduled the intermediate results that will be needed in later levels can be determined, and scheduled on the feedthrough units (the feedthrough schedule). Note that the intermediate results that have to be fed through are dependent on which add/subtracts and which multiplies were scheduled.

In order to sequence through all possible schedules, all combinations of the different add/subtract schedules, multiply schedules, and feedthrough schedules must be tried. Conceptually, this is shown in Figure 5.4. This Figure is the tree of all the possible schedules at a specific level. Each add/subtract schedule is tried with each multiply schedule, which in turn is tried with each feedthrough schedule. This tree is traversed in a depth first fashion.

Note that the size of this tree is limited by several factors. Sometimes switch connectivity prevents two inputs from getting to the same functional unit. Often a functional unit is unavailable because another operation has already been scheduled on it. A schedule

is also illegal if it causes there to be more intermediate results than there are feedthrough units. At the very first level, a schedule is valid only if a valid input operand to input register assignment is possible (i.e. the inputs can be assigned to registers in such a way that the operation and feedthrough schedules are possible). These factors all contribute to reducing the size of the tree of possible schedules.

A method is needed to sequence the individual add/subtract, multiply, and feedthrough schedules. Taking the example of an add/subtract schedule, the sequencing is accomplished by ordering the add/subtract units, and then for each operation in the add/subtract schedule assigning the units in order, so that all possible combinations of assignments of units to operations are covered. Similarly for the multiply and feedthrough schedules.

### 5.2.3 Limiting the Search

In order for the algorithm to have acceptable runtime behavior, different ways must be found to limit the search tree. The combinatorial explosion at the first level alone is enough to render the complete search infeasible. The first level is particularly bad because the complete flexibility allowed in assigning the input registers means that most of the schedules tried are possible and should be investigated. As an example, if at the first level the add/subtract schedule has 8 operations, the multiply schedule has 8 operations, and all 8 feedthrough units will be needed, then the number of schedules at the first level is over  $1.74 \times 10^{11}$ . The problem is not nearly as bad at other levels since their inputs are fixed by the outputs from the previous level and the switch connectivity limits the schedules that are possible.

The following algorithmic methods are used to prune branches of the search tree:

1. A branch-and-bound search is used [33, 37]. Once a solution has been found, a lower bound of the required remaining levels is used to prune all branches that cannot lead to a better solution than the one already found e.g. if it is known that continuing

the search along a given branch will at best require six more levels to complete the calculation, and the method that has already been found requires six or less more levels, then there is no point in continuing the search along that branch. The lower bound used is the maximum depth of an unscheduled node from an output of the DAG, times the number of level delays required to complete a single computation.

2. Branches in which no activity is occurring are eliminated. If the search gets into a state in which it continuously feeds through intermediate results without doing useful work, this is detected and corrected.
3. A lower bound of the best possible number of configurations the DAG can be mapped into is calculated, and if this optimal number is achieved, the search is stopped.

The following heuristic methods were used to guide the search and prune branches from the search tree:

1. Operations are prioritized. Each operation is assigned a priority based on its maximum distance from any output of the DAG. This priority is used to help decide which operations to schedule first.
2. Greedy scheduling of operations is used. If there is a functional unit which an operation can be scheduled on (because no other operation has been scheduled on it and because the input operands are connected via the switch) then it will be scheduled. If greedy scheduling is not done many branches are searched in which although an operation can be scheduled, it is not scheduled, but rather the operands are sent to feedthrough units. There are pathological cases where this is in fact desirable but they are rare in typical mathematical expressions. Greedy scheduling at the first level is optional because there are cases where requiring greedy scheduling at the first level makes the input register assignment impossible.
3. When scheduling the first level, the number of feedthrough schedules attempted for a

specific add/subtract and multiply schedule is artificially limited. As mentioned previously, the great flexibility at the first level is a major contribution to the complexity of the problem. Most of the time it is an incorrect assignment of the add/subtract or multiply units at the first level that prevents a better method being found. Examining all possible ways of assigning the feedthrough units when the add/subtract or multiply assignments are non-optimal is unproductive. A non-optimal feedthrough assignment at the first level is easier to correct for in subsequent levels.

4. The number of different valid schedules examined is artificially limited. A valid schedule means any schedule at any level which is looked at because it may lead to a better solution. The behavior of the algorithm is such that the depth first search usually quickly finds a method that will do the computation, though this is not usually the optimal solution. It then spends time backtracking and trying to find a better solution. Limiting the number of valid schedules examined limits the time spent in trying to improve the answer. This is similar to a time limited chess program that cuts its search short and returns the best answer found so far.

Although not currently implemented in the compiler, there is a fallback position if a method is not found for an expression. It involves breaking up the expression DAG into smaller subDAGs and scheduling the subDAGs. This is similar to the register allocation problem in which if there are not enough registers to hold all intermediate results, the fallback position is to store values in memory.

### 5.3 Other Enhancements and other Approaches

A number of other enhancements to the compiler are possible but have not been implemented. These enhancements improve performance in terms of running time and in terms of the quality of the methods found:

1. **Expression Pre-Processing.** Include a pre-processing step in which the expression graph is reorganized to minimize its depth and its resource requirements. This reorganization step would use the associative, commutative, and distributive properties to do Arithmetic-Expression Tree-Height Reduction as suggested by Kuck [20, 19]. This type of reorganization can lead to speedups of at most  $O(n/\log n)$ . Two examples of the benefits of doing this reorganization are:
  - **Accumulation Trees.** Many of the loops involve accumulating a result into a single variable by adding one partial result to the total at each iteration. To exploit the parallelism available on the RAP these are broken up into accumulation trees, as shown in Figure 5.5 in the case of a sum accumulate. In the case that intermediate results of an accumulate function are needed (e.g. each partial sum), adding redundant operations permits the use of an accumulate tree to speed up the overall computation. This is shown in Figure 5.6.
  - **Eliminate Common Subexpressions.** The compiler currently combines common subexpressions which are exact duplicates of each other. It is possible to use the distributivity and associativity of operations to find other ways to simplify the expression. For instance, Figure 5.7a is simplified by the compiler to Figure 5.7b but could be simplified to Figure 5.7c using the distributivity property of the multiply operation. This must be used with caution since eliminating all common subexpressions may not be advisable if the goal is to minimize graph depth.
2. **Redundant Operations.** The compiler can use unused functional units to do redundant computations which would improve the schedulability of the following levels. For instance, if a value is fed through two different feedthrough units it will be accessible to more functional units as an input at the next level. The same is true if an add or a multiply is done twice.

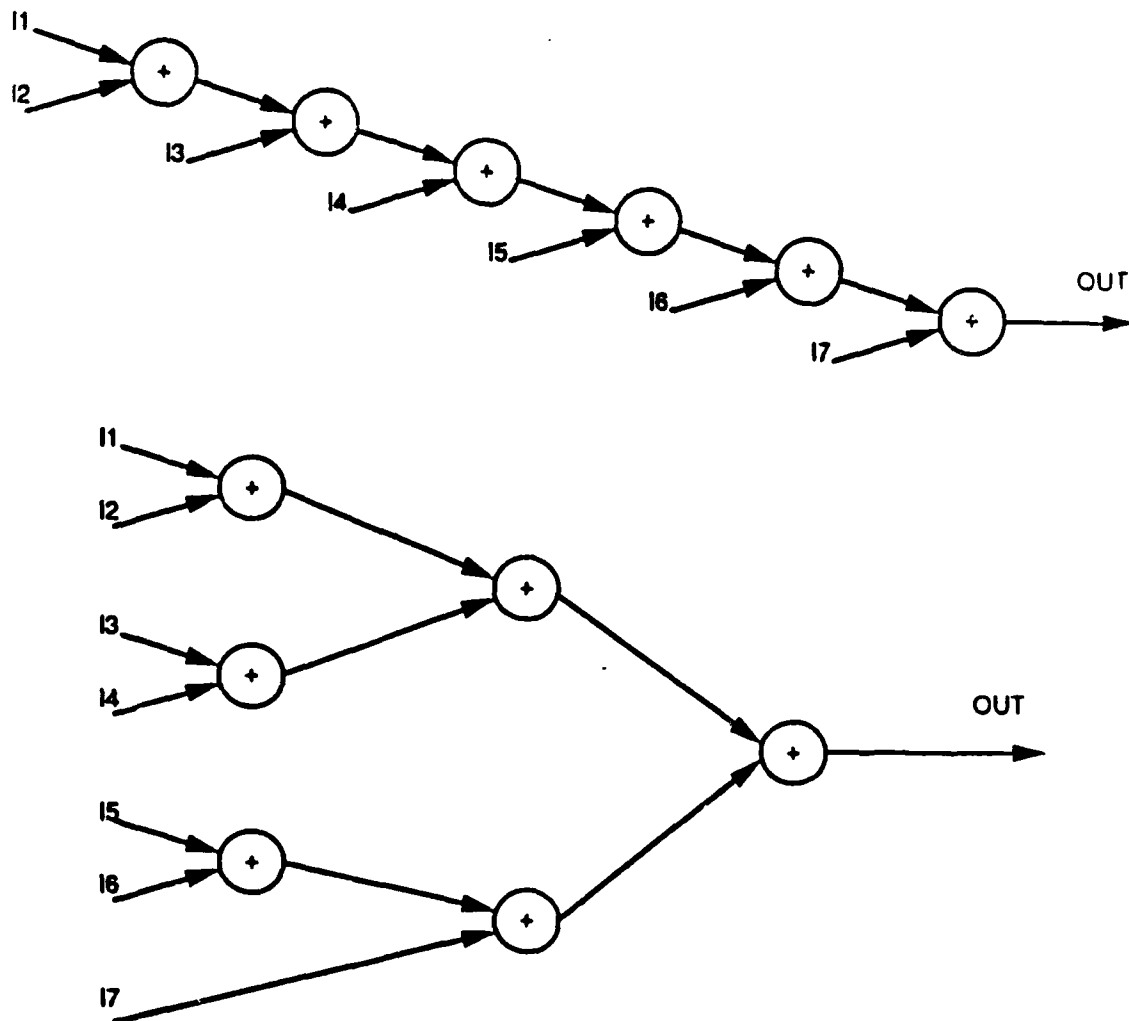


Figure 5.5: An Accumulation Tree

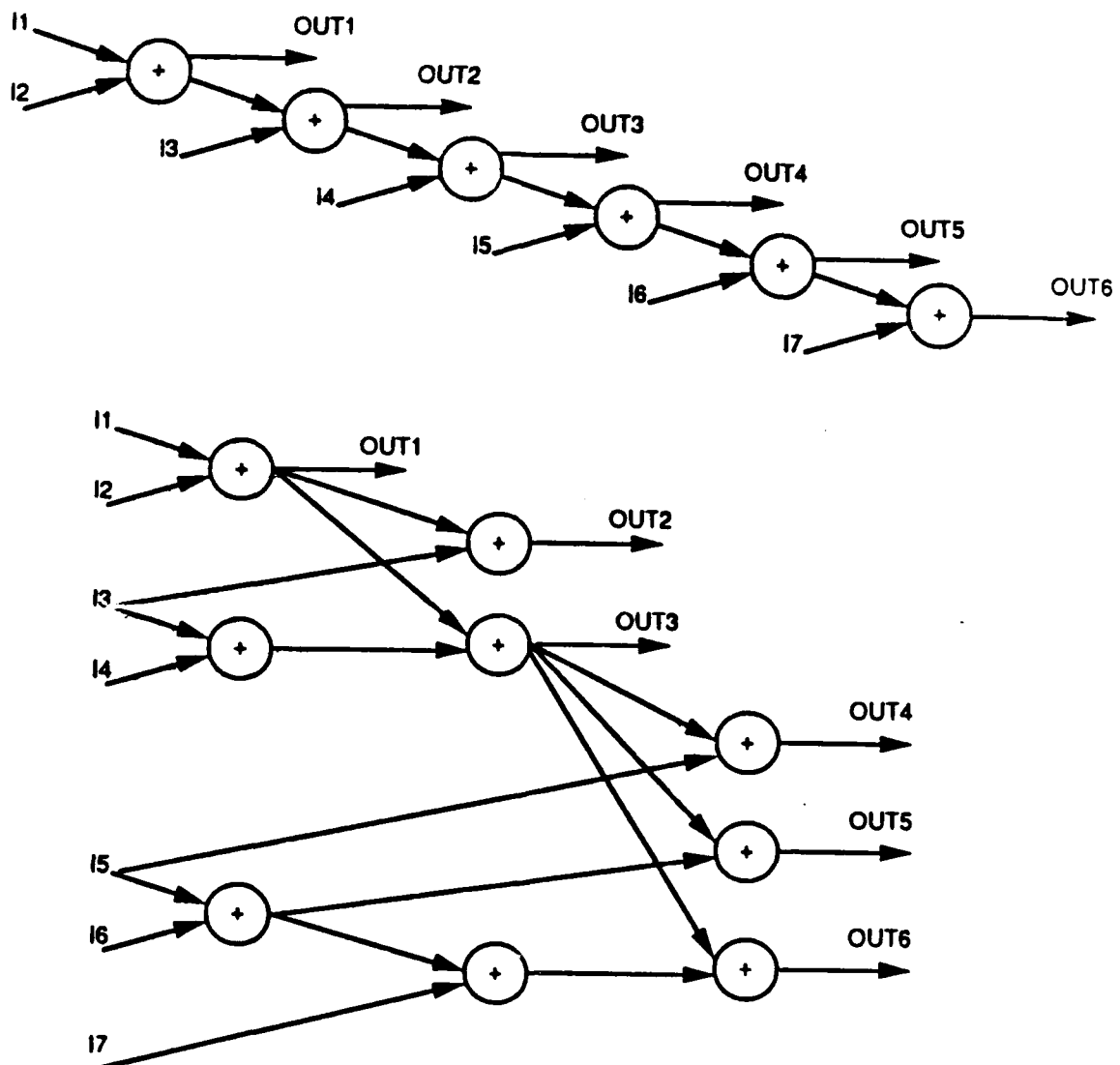


Figure 5.6: Accumulation Tree with Redundant Operations

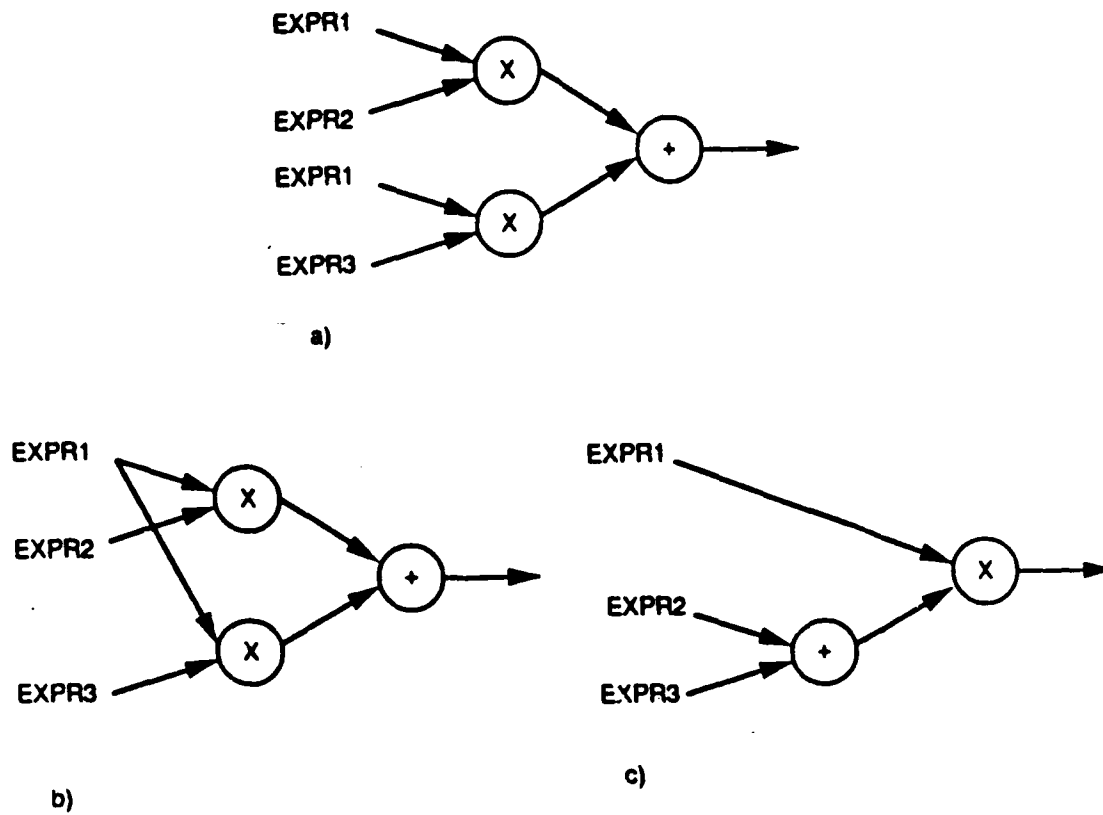


Figure 5.7: a) Unsimplified DAG b) DAG with Common Subexpressions Combined c) DAG Simplified Using the Distributivity Property

3. Redundant Input Variables. In some cases loading an input operand into more than one input register can allow operations to be scheduled that could not otherwise be scheduled. This increases the bandwidth requirements of the computation but can also increase floating-point performance.
4. If the switch/functional unit combination exhibits some form of symmetry, then this can be exploited at the first level to prevent the examining of schedules that are equivalent. This symmetry cannot be exploited at other levels because the symmetry is broken by the fact that inputs to the level are fixed by the previous level.

Completely different approaches are possible than the search algorithm used. Possibilities include:

1. Do the search backwards starting from a scheduling of the outputs.
2. Introduce critical path analysis and begin by scheduling the critical path, attempting to fit in the other computations around the already scheduled operations.
3. Rather than searching for the methods, attempt to match expressions to common expression patterns for which mappings onto the switch are known.

Further study is needed to determine the feasibility and the performance benefits, if any, of these approaches.

## 5.4 Summary

In order to use the RAP datapath, a compiler has been written that maps a mathematical expression into a series of switch configurations that perform the calculation. The compiler schedules operations on the finite resources of the RAP. The compiler allows the specification

of the RAP resources including the number and type of functional units, and the switch connectivity.

Finding an optimal schedule for the operations is an NP-complete problem. For this reason, the approach used in the compiler is to do a depth first search using algorithmic and heuristic methods to guide and limit the search. The most important of these are the prioritizing of operations based on their maximum depth from a DAG output, the greedy scheduling of operations, the use of a calculated lower bound to limit the search, and the artificial limiting of the number of schedules examined.

## Chapter 6

# Performance

*Be not swept off your feet by the vividness of the impression, but say, "Impression, wait for me a little. Let me see what you are and what you represent. Let me try you."*

— EPICTETUS, in *Discourses*, bk. II, ch. 18

*We should be careful to get out of an experience only the wisdom that is in it - and stop there; lest we be like the cat that sits on the hot stove lid. She will never sit down on a hot stove lid again - and that is well; but also she will never sit down on a cold one any more.*

— MARK TWAIN in *Following the Equator*, vol. I, *Pudd'nhead Wilson's New Calendar*, ch 11 (1897)

In this Chapter, performance of the RAP is evaluated using the simulator of Chapter 3 and the expression compiler of Chapter 5. Performance is evaluated by taking a number of mathematical expressions from the inner loops of computationally intensive programs, mapping them onto the RAP using the expression compiler, and simulating the operation of the RAP using the simulator. Performance is evaluated from two points of view: first of

all from the point of view of the bandwidth reduction achieved over the case in which no locality is exploited, and secondly from the point of view of the rate of computation achieved in Millions of Floating-Point Operations per second (MFlops). For the problems evaluated the bandwidth that has to be provided to the datapath to achieve a given performance is reduced on average by 64%. The average floating-point performance for these problems is 3.40MFlops.

This Chapter is organized as follows: in section 6.1 the assumptions made and the benchmarks used in the performance evaluation are described. Sections 6.2 and 6.3 deal with the bandwidth performance of the RAP and floating-point performance of the RAP respectively. Section 6.4 discusses what factors limit the performance achieved, and finally section 6.5 looks at various ways in which the RAP performance can be improved.

## 6.1 Evaluation Method

### 6.1.1 Assumptions

Performance figures assume a minor cycle of 12.5ns, a major cycle of 50ns, an input bandwidth of 400Mbit/sec, and an output bandwidth of 400Mbit/sec [9]. A word time is extended to five major cycles from four major cycles because one extra cycle is needed every word time to change the switch configuration. The latency of a single floating-point adder/subtractor or multiplier is three word times or 15 major cycles, and the units are assumed to be pipelined, such that peak performance is 4MFlops for each unit. The feedthrough units have a latency of one word time. The overhead in handling messages is determined by the implementation of the RAP control and is simulated by the simulator.

### 6.1.2 Benchmarks

The expressions used as benchmarks are derived from a number of numerically intensive problems involving many adds, subtracts, and multiplies. They were taken from a number of common problems such as circuit simulation, signal processing, hydrodynamics, and common vector and matrix computations. The programs used in the set of Livermore Loop benchmarks [27] are drawn upon extensively. Typically the expressions are found in the innermost loop of these programs and must be executed many times each time the program is run.

A number of techniques are used to help map the expressions onto the switch and to increase performance:

1. **Loop Unrolling.** If the innermost loop is simple, then a number of loop iterations are mapped onto the RAP at the same time.
2. **Loop Decomposition.** If the expression in the innermost loop is too large, then the expression or expressions must be broken up into several different messages to be sent to the RAP.
3. **Expression Pre-processing.** Some of the benchmark DAGs are "pre-processed" using the techniques suggested for improving the compiler in Chapter 5. The techniques used included the use of accumulation trees (including redundant operations), and redundant input variables.

The benchmarks used along with the details of how they were mapped onto the RAP are shown in table 6.1. The benchmark expressions are found in [11].

| Benchmark  | Comments                                                                                |
|------------|-----------------------------------------------------------------------------------------|
| bm-vectsum | Vector sum of two vectors of dimension 8.                                               |
| bm-accum   | Accumulate of 16 numbers in a sum.                                                      |
| bm-accum2  | Accumulate of 16 numbers in a sum and in a multiply at the same time.                   |
| bm-2x2fft  | A 2x2 FFT.                                                                              |
| bm-22x2fft | Two 2x2 FFTs computed at the same time.                                                 |
| bm-4x4fft  | A 4x4 FFT. Two extra operations introduced to allow adder units to act as feedthroughs. |
| bm-vandp   | An iterative solution to Van Der Pol's equation.                                        |
| bm-poly6   | Calculation of a polynomial of degree 6.                                                |
| bm-ids     | formula for the non-saturation drain to source current in a MOS transistor.             |
| bm-liv1    | Hydro excerpt, loop unrolled 3 times.                                                   |
| bm-liv2    | Incomplete Cholesky Conjugate Gradient, loop unrolled 3 times.                          |
| bm-liv3    | Inner Product, accumulate tree used.<br>Note: Livermore loop 6 has the same form.       |
| bm-liv4    | Banded Linear Equations, loop unrolled 5 times.                                         |
| bm-liv5    | Tri-Diagonal Elimination, loop unrolled 4 times.                                        |
| bm-liv7    | Equation of State Fragment.                                                             |
| bm-liv8    | A.D.I Integration, one expression from main loop.                                       |
| bm-liv10   | Difference Predictors, Accumulate tree and 6 redundant operations used.                 |
| bm-liv11   | First Sum, loop unrolled 10 times, Accumulate tree and 7 redundant operations used.     |
| bm-liv12   | First Difference.                                                                       |
| bm-liv18   | 2-D Explicit Hydrodynamics Fragment, one expression from main loop.                     |
| bm-liv19   | General Linear Recurrence Equation, loop unrolled 3 times.                              |
| bm-liv21   | Matrix*Matrix Product, loop unrolled 15 times, Accumulation tree used.                  |
| bm-liv23   | 2-D Implicit Hydrodynamics Fragment                                                     |

Table 6.1: Benchmarks used to Evaluate Performance

## 6.2 Bandwidth Performance

The main goal of the RAP is to use the locality found in mathematical expressions to reduce the bandwidth required to sustain high rates of floating-point computation. For the benchmarks studied the bandwidth required to sustain a given level of computation was reduced on average by 64% from the bandwidth required if no locality is exploited.

Table 6.2 summarizes the bandwidth requirements of the different benchmarks. For each problem the table shows the number of operations performed, the number of input and output words, and the number of configurations in the method. From these figures the I/O bandwidth required is calculated and compared to the maximum bandwidth that this calculation requires if no locality is exploited, to determine the reduction in bandwidth achieved by using the RAP.

The I/O bandwidth required depends on the locality inherent in the problem. For instance the vector sum benchmark, *bm-vectsum*, consists of a number of independent operations and has no locality that can be exploited by the RAP so there is no bandwidth advantage in using it (in fact if overhead is included, using the RAP is more costly). The situation is similar in the case of the *bm-liv4* and *bm-liv12* benchmarks. Except for the above mentioned three cases the bandwidth required for all the problems has been reduced to within the capacity of the network.

The bandwidth requirements also depend on how good a mapping the expression compiler found for the benchmark. The longer the method, the less bandwidth is required to sustain the computation and vice versa. However, the percentage savings of bandwidth is independent of the length of the method used.

Note that the I/O bandwidth required will increase slightly because of communication and message overhead. The percentage cost of this overhead depends on how many sets of operands are sent in a single message.

| Benchmark  | #<br>Ops | I/O Words<br>(input +<br>output) | #<br>Configs. | I/O<br>Bandwidth<br>Required | Maximum<br>Bandwidth<br>Required | %<br>Bandwidth<br>Savings |
|------------|----------|----------------------------------|---------------|------------------------------|----------------------------------|---------------------------|
| bm-vectsum | 8        | 16 + 8                           | 4             | 1536Mbit/s                   | 1536Mbit/s                       | 0%                        |
| bm-accum   | 15       | 16 + 1                           | 14            | 311Mbit/s                    | 823Mbit/s                        | 62%                       |
| bm-accum2  | 30       | 16 + 2                           | 16            | 288Mbit/s                    | 1440Mbit/s                       | 80%                       |
| bm-2x2fft  | 10       | 6 + 4                            | 12            | 213Mbit/s                    | 640Mbit/s                        | 67%                       |
| bm-22x2fft | 20       | 12 + 8                           | 17            | 301Mbit/s                    | 904Mbit/s                        | 67%                       |
| bm-4x4fft  | 36       | 15 + 8                           | 18            | 327Mbit/s                    | 1536Mbit/s                       | 79%                       |
| bm-vandp   | 20       | 8 + 4                            | 26            | 118Mbit/s                    | 591Mbit/s                        | 80%                       |
| bm-poly6   | 16       | 8 + 1                            | 23            | 100Mbit/s                    | 534Mbit/s                        | 81%                       |
| bm-ids     | 9        | 8 + 1                            | 14            | 165Mbit/s                    | 494Mbit/s                        | 67%                       |
| bm-liv1    | 15       | 10 + 3                           | 16            | 208Mbit/s                    | 720Mbit/s                        | 71%                       |
| bm-liv2    | 12       | 13 + 3                           | 10            | 410Mbit/s                    | 922Mbit/s                        | 56%                       |
| bm-liv3    | 15       | 16 + 1                           | 14            | 311Mbit/s                    | 823Mbit/s                        | 62%                       |
| bm-liv4    | 10       | 15 + 5                           | 8             | 640Mbit/s                    | 960Mbit/s                        | 33%                       |
| bm-liv5    | 8        | 9 + 4                            | 25            | 133Mbit/s                    | 246Mbit/s                        | 46%                       |
| bm-liv7    | 16       | 12 + 1                           | 24            | 139Mbit/s                    | 512Mbit/s                        | 73%                       |
| bm-liv8    | 12       | 12 + 1                           | 19            | 175Mbit/s                    | 495Mbit/s                        | 64%                       |
| bm-liv10   | 15       | 10 + 10                          | 15            | 341Mbit/s                    | 768Mbit/s                        | 56%                       |
| bm-liv11   | 17       | 12 + 10                          | 15            | 375Mbit/s                    | 870Mbit/s                        | 57%                       |
| bm-liv12   | 11       | 12 + 11                          | 5             | 1178Mbit/s                   | 1690Mbit/s                       | 30%                       |
| bm-liv18   | 13       | 12 + 1                           | 23            | 145Mbit/s                    | 434Mbit/s                        | 67%                       |
| bm-liv19   | 9        | 7 + 6                            | 30            | 111Mbit/s                    | 230Mbit/s                        | 52%                       |
| bm-liv21   | 15       | 16 + 1                           | 18            | 242Mbit/s                    | 640Mbit/s                        | 62%                       |
| bm-liv23   | 11       | 11 + 2                           | 22            | 151Mbit/s                    | 384Mbit/s                        | 61%                       |
| Average    | 15       | 12 + 4                           | 17            | 244Mbit/s                    | 682Mbit/s                        | 64%                       |

Table 6.2: RAP Bandwidth Performance

### 6.3 Floating-Point Performance

The floating-point rate attained on the benchmarks is the second key element of the RAP performance. Assuming pipelined floating-point units, with a latency of 15 major cycles, each adder/subtractor and multiplier can sustain a computation rate of 4 MFlops, giving a peak performance rate of 32 MFlops for the entire chip. For the benchmarks the average performance was 3.40MFlops and ranged from between 1.20MFlops to 8.80MFlops. On average 11% of peak power was used. The reasons for this low utilization of resources are discussed in section 6.4.

Table 6.3 summarizes the performance results for the different benchmarks. For each problem the table shows the number of operations performed, the number of switch configurations in the method, and the resulting MFlops performance. Note that for benchmarks *bm-4x4fft*, *bm-liv10*, and *bm-liv11* in which redundant operations have been introduced, the redundant operations are counted in the number of operations column, but are not counted when calculating the performance. The latency column refers to the time from when one problem instance is in the input buffer to when the complete result is in the output buffer and includes all control overhead. Table 6.4 shows the maximum achievable performance that is possible given infinite resources (i.e. unlimited switch connectivity and as many units of a specific type as are required), and the performance achievable if the same number of functional units is maintained but the switch is a complete switch. These figures are compared to the performance actually achieved: on average the limited switch used achieved 82% of the best possible performance and 88% of the performance achieved with a complete switch.

| Benchmark  | #<br>Ops. | #<br>Configs. | MFlops | Latency      |
|------------|-----------|---------------|--------|--------------|
| bm-vectsum | 8         | 4             | 8.00   | 2.90 $\mu$ s |
| bm-accum   | 15        | 14            | 4.29   | 5.05 $\mu$ s |
| bm-accum2  | 30        | 16            | 7.50   | 5.60 $\mu$ s |
| bm-2x2fft  | 10        | 12            | 3.33   | 4.20 $\mu$ s |
| bm-22x2fft | 20        | 17            | 4.71   | 5.95 $\mu$ s |
| bm-4x4fft  | 36        | 18            | 7.56   | 6.35 $\mu$ s |
| bm-vandp   | 20        | 26            | 3.10   | 7.80 $\mu$ s |
| bm-poly6   | 16        | 23            | 2.78   | 6.90 $\mu$ s |
| bm-ids     | 9         | 12            | 3.00   | 4.50 $\mu$ s |
| bm-liv1    | 15        | 16            | 3.75   | 5.35 $\mu$ s |
| bm-liv2    | 12        | 10            | 4.80   | 4.00 $\mu$ s |
| bm-liv3    | 15        | 14            | 4.29   | 5.05 $\mu$ s |
| bm-liv4    | 10        | 8             | 5.00   | 3.70 $\mu$ s |
| bm-liv5    | 8         | 25            | 1.28   | 7.60 $\mu$ s |
| bm-liv7    | 16        | 24            | 2.67   | 7.35 $\mu$ s |
| bm-liv8    | 12        | 19            | 2.53   | 6.10 $\mu$ s |
| bm-liv10   | 15        | 15            | 2.40   | 5.45 $\mu$ s |
| bm-liv11   | 17        | 15            | 2.67   | 5.55 $\mu$ s |
| bm-liv12   | 11        | 5             | 8.80   | 3.10 $\mu$ s |
| bm-liv18   | 13        | 23            | 2.26   | 7.10 $\mu$ s |
| bm-liv19   | 9         | 30            | 1.20   | 8.85 $\mu$ s |
| bm-liv21   | 15        | 18            | 3.33   | 6.05 $\mu$ s |
| bm-liv23   | 11        | 22            | 2.00   | 6.85 $\mu$ s |
| Average    | 15        | 17            | 3.40   | 5.71 $\mu$ s |

Table 6.3: RAP Floating-Point Performance

| Benchmark  | Infinite<br>Resources<br>MFlops | Complete<br>Switch<br>MFlops | Actual Switch |              |                         |
|------------|---------------------------------|------------------------------|---------------|--------------|-------------------------|
|            |                                 |                              | MFlops        | % of optimal | % of complete<br>switch |
| bm-vectsum | 10.67                           | 8.00                         | 8.00          | 75%          | 100%                    |
| bm-accum   | 5.00                            | 4.62                         | 4.29          | 86%          | 93%                     |
| bm-accum2  | 10.00                           | 9.23                         | 7.50          | 75%          | 81%                     |
| bm-2x2fft  | 4.44                            | 4.44                         | 3.33          | 75%          | 75%                     |
| bm-22x2fft | 8.89                            | 8.00                         | 4.71          | 53%          | 59%                     |
| bm-4x4fft  | 15.11                           | 9.07                         | 7.56          | 50%          | 83%                     |
| bm-vandp   | 3.33                            | 3.33                         | 3.10          | 93%          | 93%                     |
| bm-poly6   | 3.56                            | 3.56                         | 2.78          | 78%          | 78%                     |
| bm-ids     | 3.00                            | 3.00                         | 3.00          | 100%         | 100%                    |
| bm-liv1    | 5.00                            | 4.62                         | 3.75          | 75%          | 81%                     |
| bm-liv2    | 5.33                            | 4.80                         | 4.80          | 90%          | 100%                    |
| bm-liv3    | 5.00                            | 4.62                         | 4.29          | 86%          | 93%                     |
| bm-liv4    | 6.67                            | 5.71                         | 5.00          | 75%          | 88%                     |
| bm-liv5    | 1.33                            | 1.33                         | 1.28          | 96%          | 96%                     |
| bm-liv7    | 2.67                            | 2.67                         | 2.67          | 100%         | 100%                    |
| bm-liv8    | 3.20                            | 3.20                         | 2.53          | 79%          | 79%                     |
| bm-liv10   | 3.00                            | 2.78                         | 2.40          | 80%          | 87%                     |
| bm-liv11   | 3.33                            | 3.08                         | 2.67          | 80%          | 87%                     |
| bm-liv12   | 14.66                           | 8.80                         | 8.80          | 60%          | 100%                    |
| bm-liv18   | 2.88                            | 2.88                         | 2.26          | 78%          | 78%                     |
| bm-liv19   | 1.33                            | 1.33                         | 1.20          | 90%          | 90%                     |
| bm-liv21   | 4.00                            | 3.75                         | 3.33          | 83%          | 89%                     |
| bm-liv23   | 2.10                            | 2.00                         | 2.00          | 95%          | 100%                    |
| Average    | 4.13                            | 3.88                         | 3.40          | 82%          | 88%                     |

Table 6.4: Comparison of Actual Performance to Ideal Performance

## 6.4 Limits on Performance

### 6.4.1 Limits on Bandwidth Performance

The amount by which the RAP is able to reduce bandwidth requirements is limited by two factors: the structure of the computation to be performed, and the current control scheme which does not allow exploitation of locality outside of the calculation of an individual expression.

The structure of the computation determines how much locality is present in the expression to be exploited by the RAP. If the computation calculates many intermediate results that get combined to give the final results, then the RAP will be very effective in reducing the overall bandwidth requirements. If on the other hand the expression is a series of independent adds, subtracts, and multiplies, the RAP will not be as effective.

In the RAP, locality is exploited within an expression but not between expressions. All calculations are modular in that the input operands are sent to the RAP, the calculation is performed, and the results are sent to a predetermined destination. No provision is made for different calculations to share their input variables or their output results. For instance, an expression might use the same constants all the time, but the RAP requires that each time the expression is calculated the constants be sent as input variables. As another example, consider the case of accumulating 256 numbers in a sum. This is done as follows: the RAP first does 16 accumulates of 16 numbers, these 16 intermediate results are gathered together at an MDP, which then sends another message to the RAP to do the final accumulate. If the RAP had mechanisms for storing its outputs results and using them as inputs later, then the bandwidth costs of sending the intermediate results to the MDP and receiving them back would be eliminated.

### 6.4.2 Limits on Floating-Point Performance

The limitations on the floating-point performance achievable by the RAP are due to the structure of the computation to be performed, the latency of the AUs, and to the resource limitations of the RAP.

First of all, the computation is limited by the data dependencies of the problem: operations which take inputs from other operations cannot be started until the results from these other operations are available. Even with infinite resources the maximum depth of the expression DAG that represents the computation determines the minimum number of configurations that will be required to complete the computation. Furthermore, most common computations have a binary treelike structure, which have lots of operations to do at the beginning (at the leaves of the tree), but have many less operations to do as the computation progresses towards the final result (the root of the tree). This means that typically the AUs will be used heavily at the beginning of a computation but will be mostly idle towards the end of the computation (note that there are exceptions to this generalization: The FFT expressions have as many operations at the last level of their DAG as at the intermediate levels). These data dependency limitations prevent the RAP from being used at even close to peak performance: the best performance that can be achieved for the different benchmarks is given in the last column of Table 6.3.

The three word latency of the AUs compounds the data dependency problem: the computation can never complete faster than the maximum depth of the DAG times the latency of the AUs. Reducing the AU latency will lead to increased performance.

Resource constraints also limit performance. RAP resources include the number and distribution of each type of functional unit, and the switch interconnectivity. Often, the computation would proceed faster if there were only more adders or multipliers available, or if the switch had extra connections allowing inputs to reach the same functional unit, or if extra input registers or feedthrough units were available. Comparing the MFlops

actually achieved to maximum possible achievable in Table 6.4 provides a figure of merit for the RAP resources and the expression compiler. For the benchmarks used the RAP achieved 82% of the maximum possible performance for that problem, and 88% of the performance achievable if the switch is complete. As is discussed in section 6.5, changing the resource configuration may lead to better performance. For the problems considered, allowing unlimited switch resources (i.e. having a complete switch) means that the RAP would achieve 91% of the maximum achievable performance. This means that if the compiler is assumed to generate the best method for the given resources (this is a good approximation for the switch considered) then 53% of the performance degradation from the ideal case can be attributed to the limited switch connectivity, and the remaining 47% to the limited number of functional units. This suggests that the limited switch and the limited number of functional units are about equally responsible for the performance degradation from the ideal case.

## 6.5 Improving Performance

### 6.5.1 Finding Better Methods

Throughout the discussion on mapping expressions to switch configurations in Chapter 5, a number of ways of to improve the methods found were suggested, and it is important to see how these relate to the factors that limit the performance as discussed in section 6.4.

Two mechanisms can be used to increase resource utilization and limit the effect of data dependencies:

1. Map more than one expression onto the switch at once so that utilization of the functional units increases. In a number of the benchmarks studied, this was done by unrolling loops (see Table 6.3) but it can also be done by mapping completely

independent expressions onto the switch. The extent to which this mechanism can be used is limited by the number of input registers and functional units available.

2. The DAG that represents the computation can be rearranged to minimize its depth, as discussed in Chapter 5.

Mechanisms also exist which can reduce the effects of resource constraints:

1. Idle functional units can be used to do redundant operations and feedthroughs which will improve the schedulability of the following levels. If a result of a given node is available in more than one place, the incomplete connectivity of the switch can be masked more effectively.
2. Input operands can be loaded into more than one input register if some of the input registers are unused. The effect is the same as doing redundant operations on unused functional units. Note that under the current control scheme this means input bandwidth requirements will also increase.

### 6.5.2 Using Different Resource Configurations

There are two dimensions of the RAP datapath that have an effect on the RAP performance: the number of each type of functional unit, and the switch connectivity.

The optimal number of each type of functional units depends on the problem. Currently, there are four add/subtract units and four multiply units. However, many problems have more adds than multiplies so that having more add/subtract units than multiply units may be advantageous. This is highly problem dependent, and a more careful study is needed to determine whether an asymmetry in the number of functional units is desirable. This choice is also limited by practical considerations, such as how much silicon area each type

of functional unit requires (e.g. it is easier to add another feedthrough unit than it is to add an add/subtract unit or a multiply unit).

The switch connectivity is a variable which has an effect on performance. For instance, the methods found by the compiler for the benchmarks could in many cases be optimal if only the switch connectivity had been different. The advantage of the current switch is that it is symmetric and offers equal loading on all the input lines. The choice of this switch however was somewhat arbitrary, and the question that must be answered is whether there are certain characteristics that are desirable in the switch. In trying to map expressions onto the switch, two desirable properties of the switch have become evident:

1. **Embedded Trees.** Since the expressions computed on the RAP typically have the shape of binary trees, being able to map trees onto the switch is a desirable property. A simple example of this is the case of the accumulate benchmark *bm-accum*, that takes 14 configurations rather than the optimal 13 configurations (13 is optimal in the case that only four adders are present, whereas 12 is optimal if unlimited resources are assumed). The extra configuration is required because at one point in the computation values have to go through feedthroughs so that the two operands can both reach the same adder. If the accumulation tree could be mapped directly onto the switch without the extra feedthrough stage, then the performance would be improved.
2. **Wide Fanout.** It is desirable to have some inputs that are fanned out to many of the functional units. In many cases a result is needed in more than one place and must be fanned out to multiple units. In the case of the switch used for performance evaluation, a given output can only reach two adders and two multipliers. If a third add or multiply using that value is ready to begin, it will be delayed and the value will have to go through a feedthrough until an adder it can reach is available.

In order to experiment with improving performance by changing the switch configuration, the switch of Figure 6.1 was used with the compiler of Chapter 5. This switch looks

rather irregular but it has the following interesting properties: the inputs 0, 1, and 2 fanout to all the AU units, and most trees can be directly mapped into a succession of switch configurations that do not include unnecessary feedthroughs. Unfortunately, this irregular switch makes it so that the compiler has to search a lot longer to find a good solution, because although the optimal solution is possible, there is usually only one optimal solution. This argues in favor having a more complete switch or a compiler that is more aware of switch structure: the search could be directed towards matching common expression patterns for which the optimal sequence of switch configurations is known.

### 6.5.3 Pipelining RAPs

RAPs can be used in a pipeline in which each RAP does part of the computation and passes intermediate results to the next RAP for the next part of the computation. Pipelining increases performance by allowing more than one RAP to work on the same problem. For example, if one RAP does the first half of the configurations in a method and then passes intermediate results onto the next RAP which executes the second half of the configurations of the method, then overall performance can be increased by a factor of two. As in any pipeline, balancing the stages of the pipeline is important: each RAP in the pipeline should have the same amount of work to do i.e. the same number of configurations. Also, the network is an important part of the pipeline since between every two RAP stages there is a network stage. Matching the network speed to the RAP speed is an important consideration. To achieve balance, each method should have sufficient configurations to keep the RAP busy without overloading the network, and few enough for the RAP to keep up with the network. Table 6.5 shows how many configurations in a method would be ideal, given the number of input operands. The minimum number of configurations that does useful work is three, the latency of one operation. Note these figures assume an unloaded network: if there is other traffic on the network the I/O bandwidth will decrease, the time for a complete set of operands to arrive, and the ideal number of configurations per method will both increase.

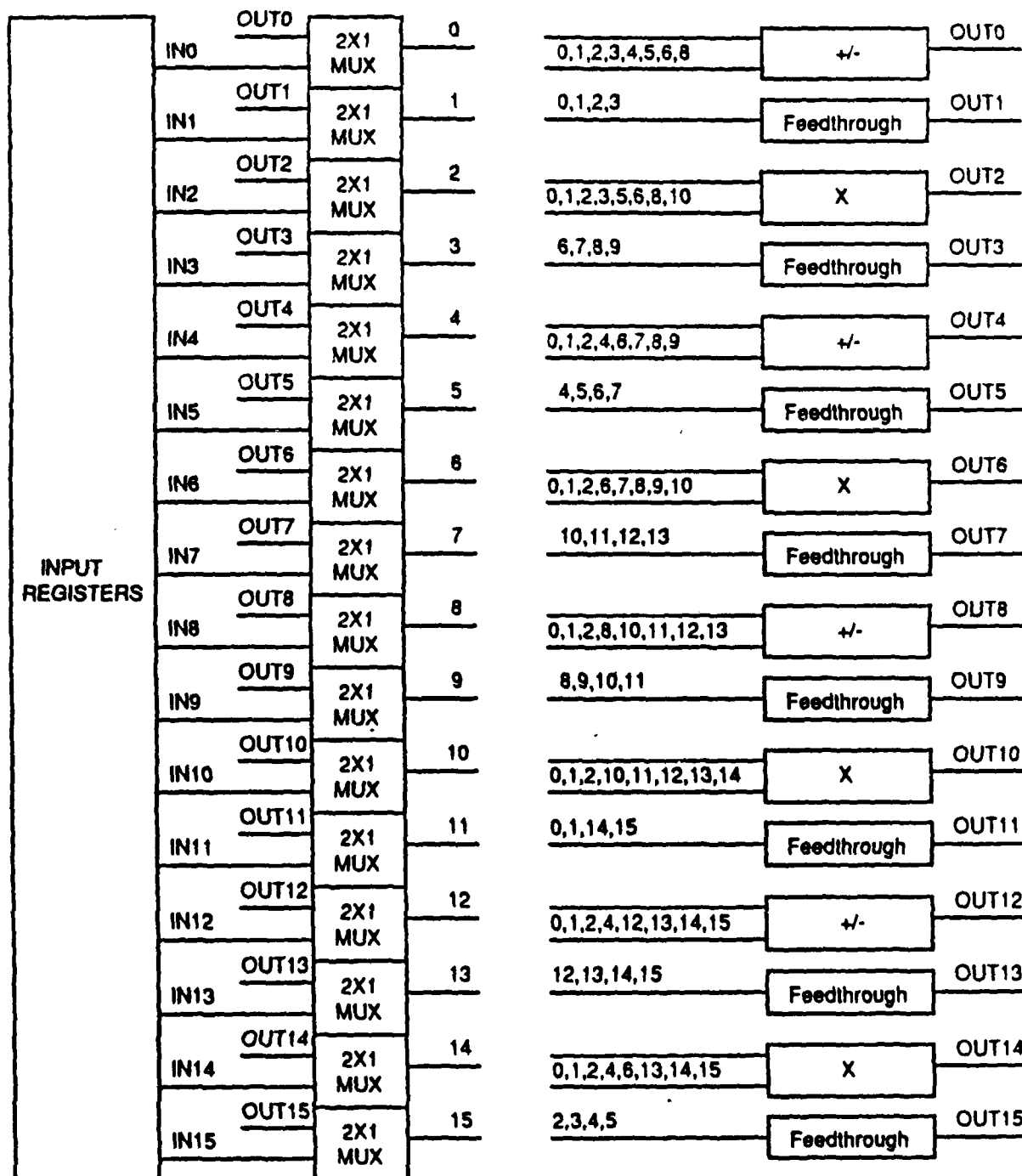


Figure 6.1: An Alternative Switch

| # Operands | Time for a set of operands to arrive | Best Number of Configurations |
|------------|--------------------------------------|-------------------------------|
| 1          | 160ns                                | 3                             |
| 2          | 320ns                                | 3                             |
| 3          | 480ns                                | 3                             |
| 4          | 640ns                                | 3                             |
| 5          | 800ns                                | 3                             |
| 6          | 960ns                                | 4                             |
| 7          | 1.12 $\mu$ s                         | 4                             |
| 8          | 1.28 $\mu$ s                         | 5                             |
| 9          | 1.44 $\mu$ s                         | 6                             |
| 10         | 1.60 $\mu$ s                         | 6                             |
| 11         | 1.76 $\mu$ s                         | 7                             |
| 12         | 1.92 $\mu$ s                         | 8                             |
| 13         | 2.08 $\mu$ s                         | 8                             |
| 14         | 2.24 $\mu$ s                         | 9                             |
| 15         | 2.40 $\mu$ s                         | 10                            |
| 16         | 2.56 $\mu$ s                         | 10                            |

Table 6.5: Optimum # of Configurations per Method vs. Number of Input Operands

For those computations which have long methods, dividing the computation into a pipeline over several RAPs is advantageous. This requires finding the right place to break up a method. Any mismatch in speed between the network and the RAP can be somewhat compensated for by the input and output buffers, but in the worst case may back up the network.

#### 6.5.4 Reorganizing Control and Operation of the Data Path

Reorganizing how the RAP datapath is controlled can lead to both improved bandwidth performance, and to improved floating-point performance. This is achieved however at the cost of increased complexity in the control. The extra performance gained by these schemes must be carefully measured against the extra cost in complexity.

### **Decreasing Bandwidth Requirements**

A very simple idea for reducing the bandwidth required for many expressions is to include constants in the expression as part of the method definition. This means that the constants do not have to be sent as input operands every time the method is calculated.

A more difficult idea to implement is to give the RAP a more general addressing capability that would allow it to store the input operands and the results of its computations in memory, and to combine these stored values in subsequent computations. This offers the possibility of further decreasing the off chip bandwidth but would require a substantial increase in control complexity.

### **Increasing Floating-Point Performance**

The control scheme of the RAP datapath is currently set up in a way that minimizes complexity. Problems are loaded one at a time, and only one problem can be in the functional units at a time. Allowing only one problem to be calculating at a time means that for the common tree structure, lots of work is done at the beginning of the computation (at the leaves of the tree) but as computation progresses utilization of the functional units decreases, and the floating-point performance decreases.

Performance can be increased by using different control schemes that relax the constraint of having only one problem calculating at once. If the feedthroughs are extended to have three word delays, and the datapath is considered as three different machines that can calculate three different problems, then performance can be increased by a factor of three if there is sufficient bandwidth. In this scheme the overall performance increases, but the speed of calculation of one particular expression decreases. Alternatively, the methods could be compiled so that when doing calculations on many sets of operands, computation on one set of operands begins before the calculation on the previous set completes. This is similar to software pipelining in VLIW machines [21] in which iterations of a loop in a program are

initiated at constant intervals, before the preceding iterations complete. In the case of the tree structure, this means that another tree begins to calculate just as the previous tree is beginning to have less work to do, and thus resource utilization increases.

## 6.6 Summary

There are two aspects of the RAP performance that are important: the bandwidth required to sustain a given level of computation, and the floating-point performance achieved. 23 benchmarks consisting in mathematical expressions taken from the inner loops of computationally intensive programs were used to evaluate performance.

The RAP reduces I/O bandwidth that must be provided to the datapath by 64% when compared to the bandwidth required if no locality is exploited. The amount bandwidth is reduced by is limited by two factors: the locality present in the benchmark, and the fact that the RAP does not allow sharing of input variables or results between different expressions.

The RAP achieves an average floating-point performance of 3.40MFlops over the 23 benchmarks. This performance corresponds 82% of the performance achievable if infinite resources were available, and to 88% of the performance achievable if the number of units is limited but a complete switch is used. The floating-point performance is limited by the amount of parallelism present in the calculation, by the AU latency, and by resource constraints.

Numerous schemes can be envisioned for improving performance. Bandwidth performance can be improved by allowing constants to be included as part of the RAP, and by allowing the RAP to use on chip memory to exploit locality beyond the expression level. Floating-point performance can be improved in several ways. Better methods can be found by improving the compiler, and by improving the resource configuration to allow good map-

pings to be found more easily. The RAP control structure could be changed to allow more than one problem to be computing at one time. Also, floating-point performance can be increased at the system level by pipelining RAPs.

## Chapter 7

## Conclusion

*No thing great is created suddenly, any more than  
a bunch of grapes or a fig. If you tell me that you  
desire a fig, I answer you that there must be time.  
Let it first blossom, then bear fruit, then ripen.*

— EPICTETUS, in *Discourses*, bk. I, ch. 15

*Perhaps someday it will be pleasant to remember even this.*

— VIRGIL in *Aeneid*, bk. I, l. 203

*Now is the time for drinking, now the time  
to beat the earth with unfettered foot.*

— HORACE in *Odes*, bk. I, ode xxxvii, l. 1 (23 B.C.)

## 7.1 Summary

The main problem in achieving high performance floating-point is supplying the I/O bandwidth, both on and off chip, necessary to keep fast floating-point circuits and datapaths busy. This Thesis describes and evaluates the Reconfigurable Arithmetic Processor (RAP) architecture, which is designed to substantially reduce the bandwidth required to do high performance floating-point.

The RAP uses three main mechanisms to reduce bandwidth: use of locality, serial arithmetic, and a reconfigurable datapath. First, it exploits the locality inherent in mathematical formulas by calculating complete arithmetic expressions without storing intermediate results in memory or in register banks. This eliminates the bandwidth costs associated with storing and retrieving intermediate results. Second, it uses serial arithmetic which allows area efficient implementations of floating-point arithmetic. This area efficiency allows several floating-point units to be put on a single chip and these units can be run in parallel to achieve high performance. Third, the RAP uses the idea of a reconfigurable datapath that permits the routing of intermediate results between functional units using a switch. The calculation of a mathematical expression involves sequencing the switch through different configurations, thus routing operands and intermediate results to appropriate functional units.

The RAP is designed to fit into the J-Machine [8], a message passing multiprocessor system. The RAP is controlled by three simple messages, which allow it to be used in this system. These messages allow mathematical expression "programs" known as methods to be stored on the RAP, and provide a convenient mechanism for invoking these methods on different data inputs. A mechanism is also provided to permit the pipelining of RAPs, forking operations, and merging operations.

Two important aspects of the RAP hardware design are investigated in this Thesis: the design of the control logic and the design of the serial floating-point units. The control

is conveniently divided into input control, switch control, output control, and network interface control. The simple flow charts for these logic blocks can easily be implemented using random logic or small PLAs.

Two types of serial floating-point functional units are used in the RAP, one a floating-point adder/subtractor, the other a floating-point multiplier. The implementation of these units is based on doing 4-bit serial arithmetic. Doing 4-bit arithmetic allows more efficient use of the logic than 1-bit or 2-bit serial arithmetic, and can be implemented with simple extensions of 2-bit algorithms. In particular, the mantissa multiply portion of the floating-point multiplier is based on Modified Booth encoding, a technique that has been used to implement two-bit serial fixed point arithmetic [23]. The most critical circuit in these units is the 4-bit adder used, due to the long carry delay. SPICE simulations of different 4-bit adders indicate that it will be possible to run these units at 80MHz. The current arithmetic unit designs have a performance of 1.57MFlops, and an improved design that allows problems to be pipelined one immediately following the other would have a performance of 4.70MFlops.

An expression compiler is used to map mathematical expressions onto the RAP datapath. This compiler takes as input a list of mathematical expressions that contain several adds, subtracts, and multiplies. It then outputs a series of switch configurations that will route operands and intermediate results to functional units that will perform the computation. This compiler uses a depth first search, trying to assign operations to functional units, while taking into account data dependencies, the number and characteristics (i.e. the functions each unit can perform and their latency) of the functional units available, and the characteristics of the switch. Algorithmic and heuristic methods are used to limit and guide the search.

The performance of the RAP has been evaluated in terms of the bandwidth savings realized, and in terms of the floating-point performance achieved. The RAP evaluated contains four add/subtract units, four multiply units, and eight feedthrough units (feedthrough units are simple delay elements used to align operands in time and space when two operands are

not ready at the same time, or cannot reach the same arithmetic unit). The expression compiler mapped 23 benchmark expressions onto a switch that has less than half the connectivity of a complete switch. Average bandwidth savings is 64% over the case where no locality is exploited. Average floating-point performance for these problems is 3.40MFlops, or 11% of the 32MFlop peak rate. This corresponds to 82% of the floating-point performance achievable if infinite resources were used (infinite resources corresponds to an unlimited amount of functional units and complete switch connectivity), and 88% of the performance achievable if a complete switch is used. These figures justify the use of an incomplete switch.

## 7.2 Future Work

This Thesis deals with the full spectrum of problems encountered in the RAP design, from the hardware implementation, to compilation and system level issues. As a result of this broad approach, it was not possible to carry out the analysis of all the different issues to great depth. Much further work is needed on all aspects the design.

Bit-serial implementations of floating-point arithmetic is an area of research unto itself. Different algorithms for doing serial floating-point are possible, and a comprehensive study of the alternatives is needed, with emphasis on achieving high performance and low latency. In particular, the use of redundant number representations and digit on-line algorithms [16, 29, 35] should be compared to the more conventional two's complement arithmetic approach. An important issue that has not been dealt with in this Thesis is the serial implementation of common floating-point operations such as divide and square root. Algorithms for these operations in the bit-parallel world are of the compare-shift type, such as in the SRT division algorithm, or of the iterative type where repeated multiplication is used to converge to the result [18]. What approach is best in the bit-serial world is an interesting research issue.

Investigating new ways to improve the floating-point performance and bandwidth per-

formance is an important area for further study. Currently the RAP floating-point performance is limited principally by the parallelism present within one mathematical expression. Finding what mechanisms are needed to allow the RAP to efficiently exploit parallelism between mathematical expressions is the key to increasing the floating-point performance. Exploiting the parallelism between expressions in turn raises many compiler level issues. Exploiting local memory to further reduce bandwidth requirements is likely to lead to significant off chip bandwidth reductions. One could envision having a RAP-like datapath on a chip with a subsidiary register file. The RAP datapath would reduce the on chip I/O bandwidth so that the register file would not have to be multi-ported. At the same time, the register file could be used to store results coming out of the datapath. This would reduce off chip I/O since the results could be stored on chip for later use.

Finally, many system level issues remain to be addressed. How does one break up problems for the RAP? How does one decide how to distribute computation over different RAPs in a multiprocessor system like the J-Machine? Should RAP pipelining be used or will this put too large a burden on the communication network? These problems are part of the larger problem of resource management and utilization that must be dealt with in all computer systems.

The idea of having several serial floating-point units on a chip connected with a switch, represents a flexible and efficient alternative to bit parallel arithmetic and a register file. The ideas found in this Thesis, including high performance serial floating-point implementations, the techniques for exposing and exploiting parallelism in arithmetic computation, and the methods for reducing I/O bandwidth, can be developed much further. As this development occurs, the RAP and its variations will become increasingly attractive as solutions to the problem of achieving high performance floating-point.

## Appendix A

# 4-Bit Adder Simulation and Layout

This appendix contains the SPICE models and the schematics used in the simulation of different 4-bit adders, as well as the layout for the basic adder cells used to estimate area.

## A.1 Simulation

The following SPICE transistor parameter models were used. These are taken from the VTI 2 $\mu$ m process:

```
.TEMP 110
*Slow NMOS, Slow PMOS models, V-CMOS process.
.MODEL NMOS NMOS
+LEVEL=2 VT0=0.75 TOX=0.0400U NSUB=3.50E16
+XJ=0.15U LD=0.20U U0=650 VMAX=5.1E4
+UCRIT=0.62E5 UEXP=0.125
+PB=0.80 NEFF=4.0 DELTA=1.4
+CGS0=195.P CGD0=195.P CJ=195.U MJ=0.76
+CJSW=50Q.P MJSW=0.30
+RSH=38
.MODEL PMOS PMOS
+LEVEL=2 VT0=-0.75 TOX=0.0400U NSUB=6.0E15
+XJ=0.05U LD=0.20U U0=255 VMAX=3.0E4
+UCRIT=0.86E5 UEXP=0.29
+PB=0.80 NEFF=2.65 DELTA=1.0
+CGS0=190.P CGD0=190.P CJ=250.U MJ=0.535
+CJSW=350.P MJSW=0.34
+RSH=110
```

The four adders simulated are:

1. ADDER1: Precharged Manchester carry chain.
2. ADDER2: Precharged Manchester carry chain with positive feedback pulldown circuitry.
3. ADDER3: Lookahead adder using domino logic.
4. ADDER4: Ripple carry adder with optimized carry path.

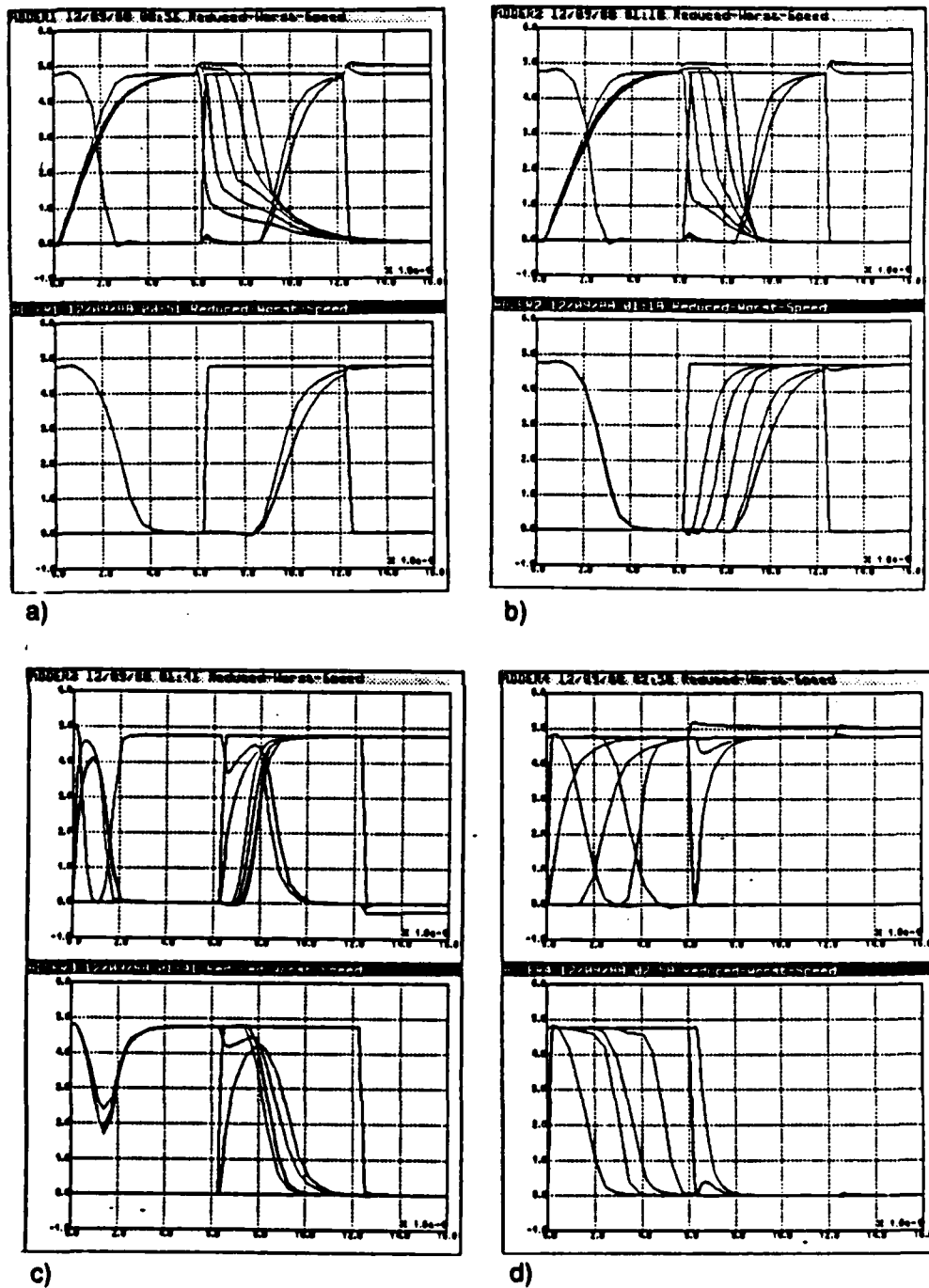
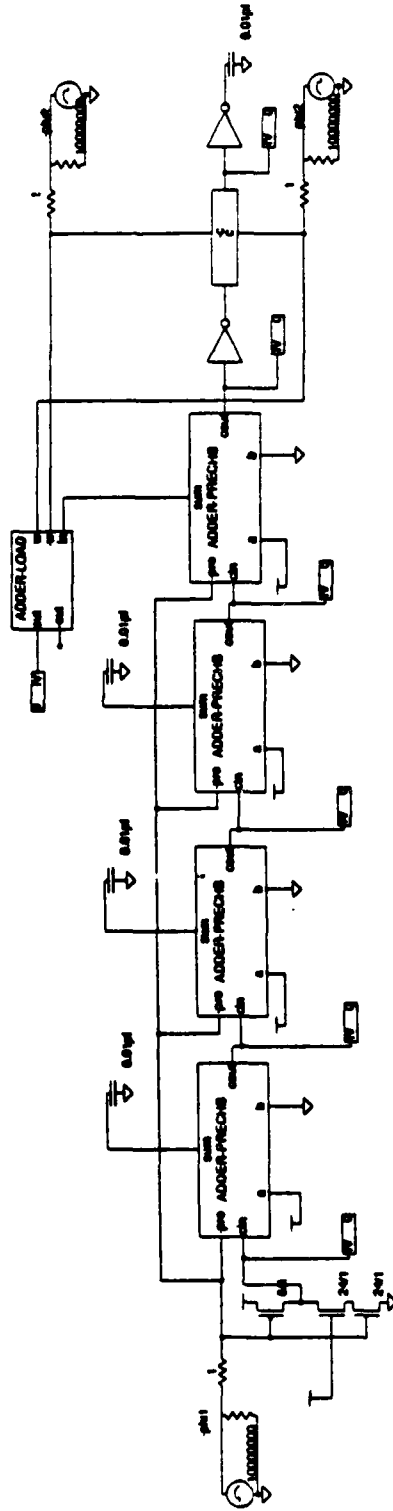


Figure A.1: SPICE Plots for Different 4-Bit Adders a) Manchester Carry Adder b) Manchester Carry Adder with Positive Feedback Pulldown Circuitry c) Carry Lookahead Adder d) Ripple Carry Adder

Figure A.1 shows the SPICE output waveforms for the four adders. For each adder two plots are shown. The first is a plot of the carry signal at each stage, including the final storage stage where the final carry out is latched. The second is a plot of the sum output at each stage, as well as the output of the register in which the final bit is stored. For reference, one of the clock signals is plotted in each case. Phase two is plotted for the first three precharged adders, while phase one is plotted for the ripple adder. Delays were measured from the 2.5V mark. Since the storage node of the output carry load is non-restoring, care must be taken that the voltage reached is above the p-transistor threshold ( $\simeq 4V$ ) for a high going voltage, and below the n-transistor threshold ( $\simeq 1V$ ) for a low going voltage, before the falling edge of the clock.

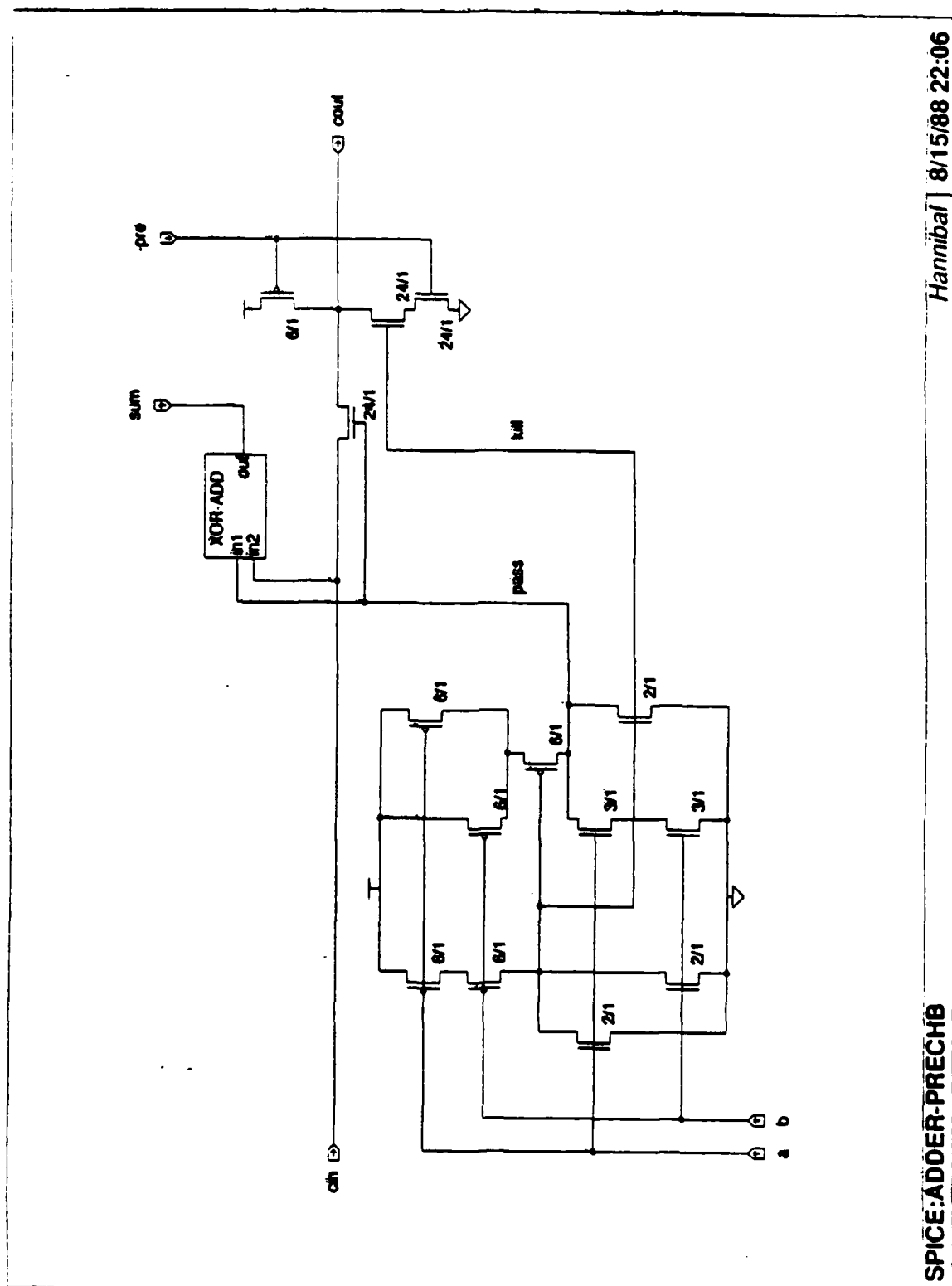
## A.2 4-bit Adder Schematics

This section contains the schematics of the 4-bit adders that were used in the simulation.

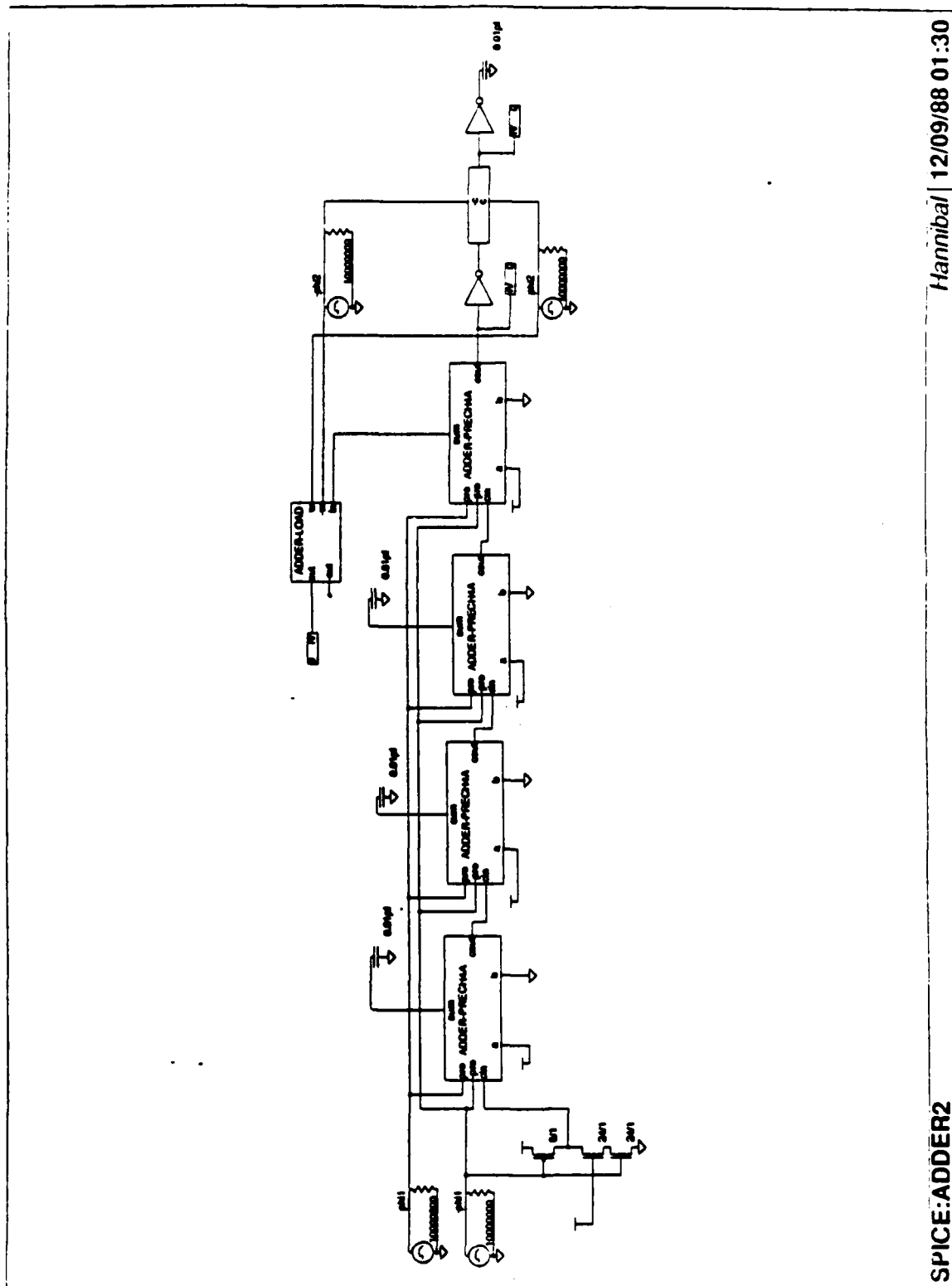


SPICE: ADDER1

Hannibal | 12/09/88 00:54

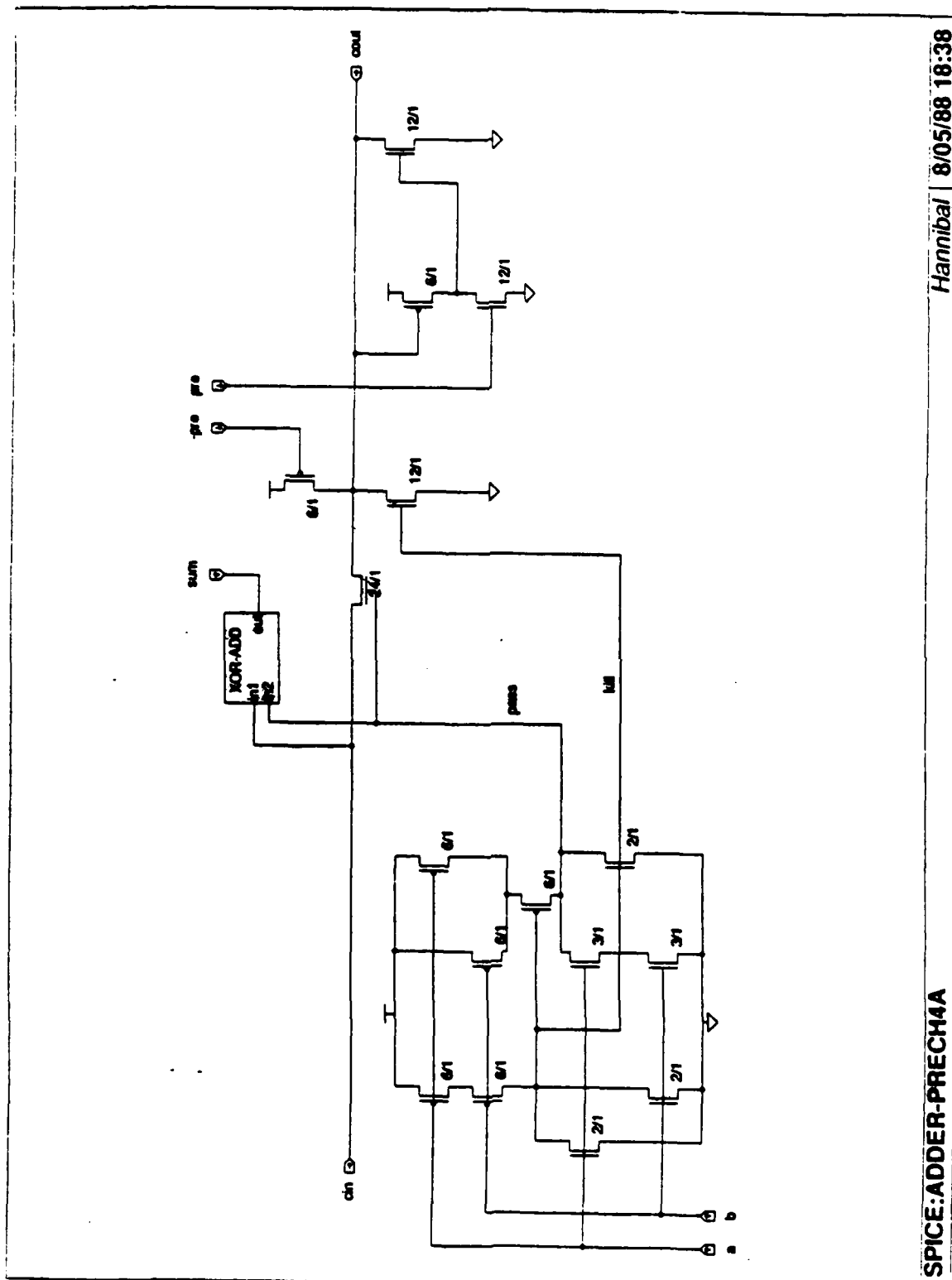


SPICE:ADDER-PRECHB Hannibal 8/15/88 22:06



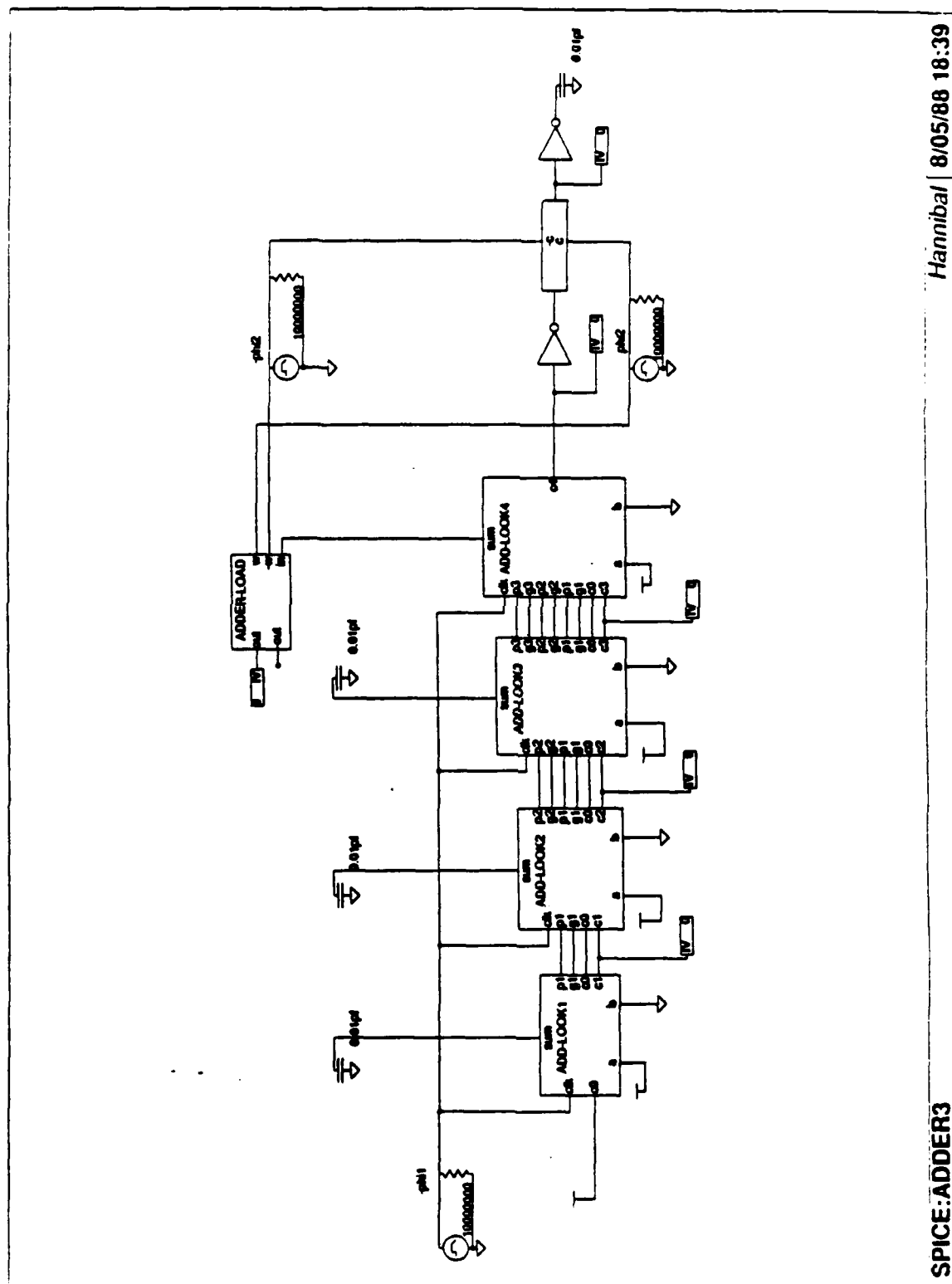
SPICE:ADDER2

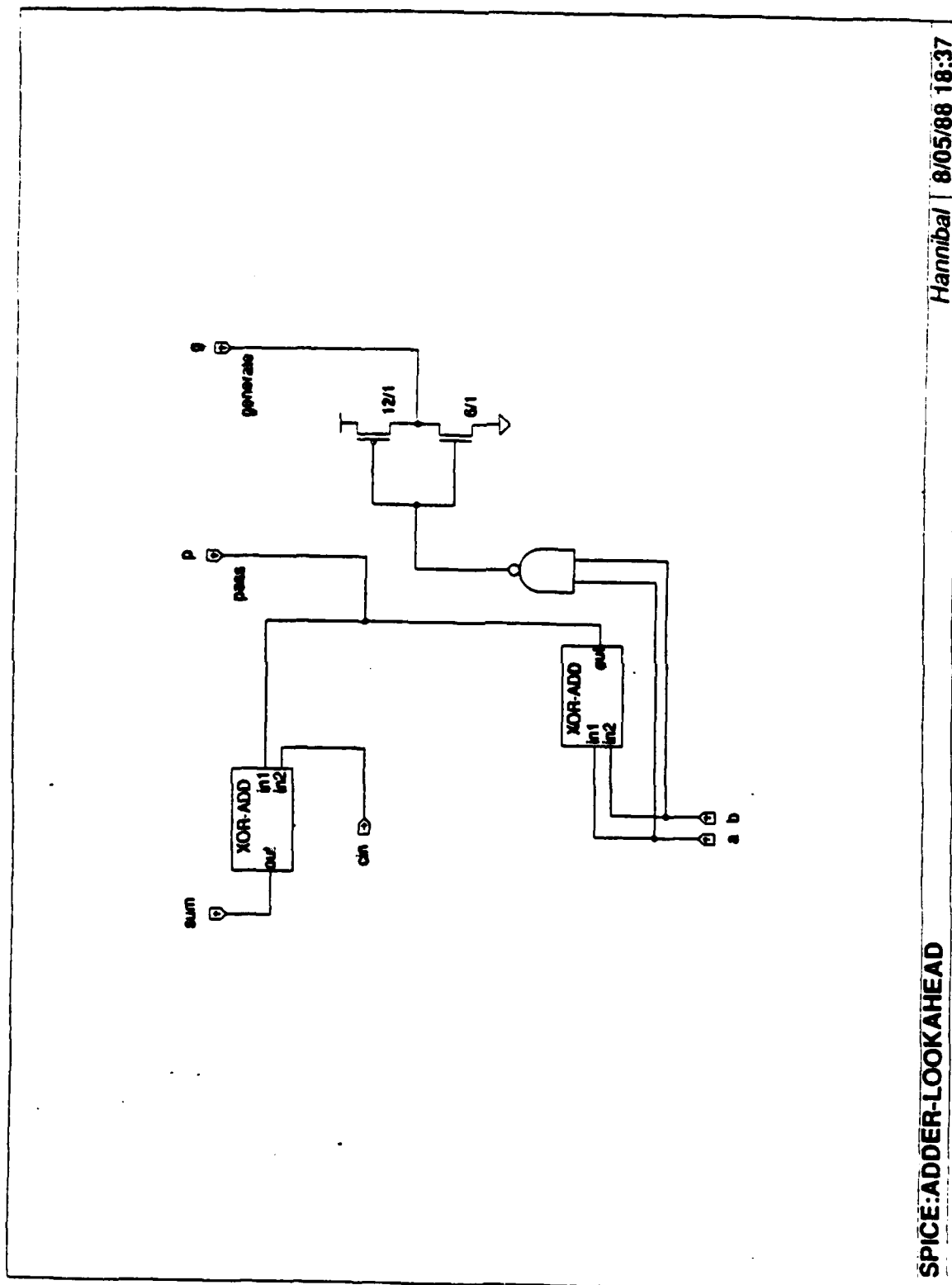
Hannibal | 12/09/88 01:30

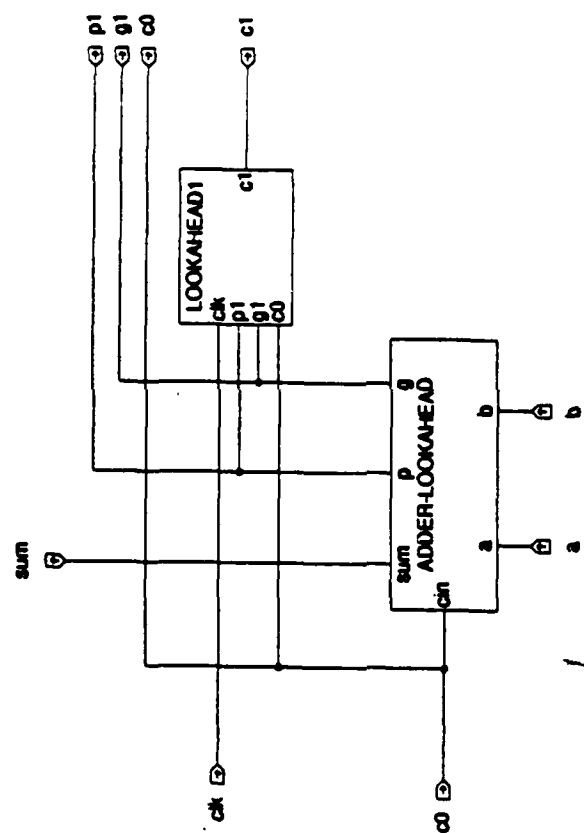


Hannibal 8/05/88 18:38

SPICE:ADDER-PRECH4A

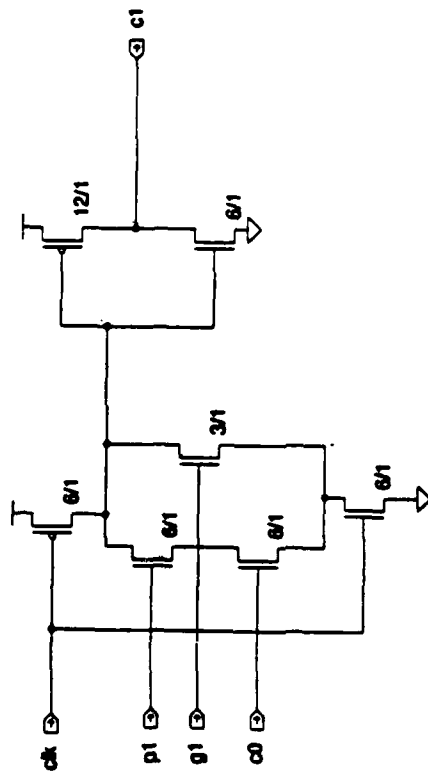






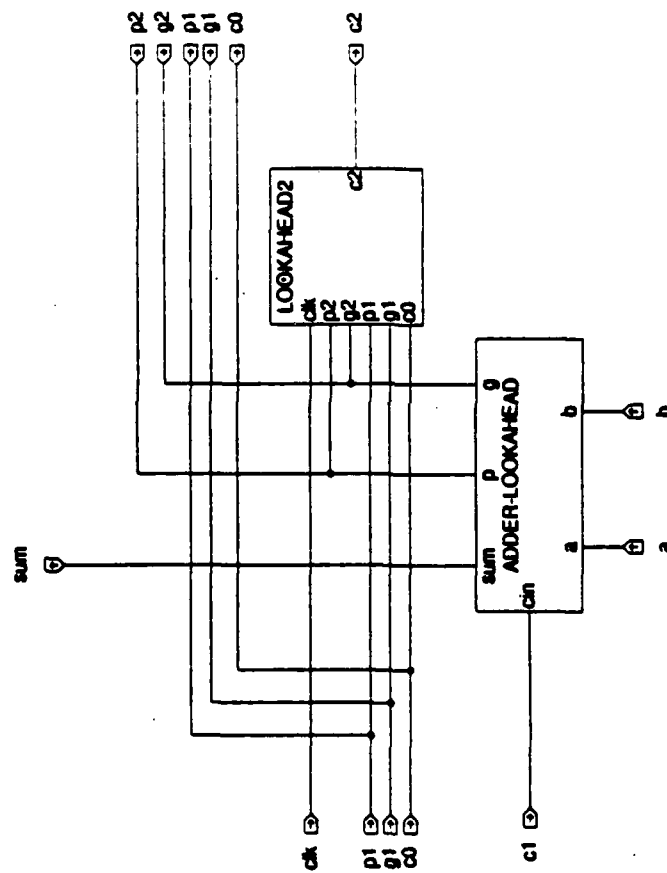
SPICE:ADD-LOOK1

Hannibal | 8/05/88 18:37



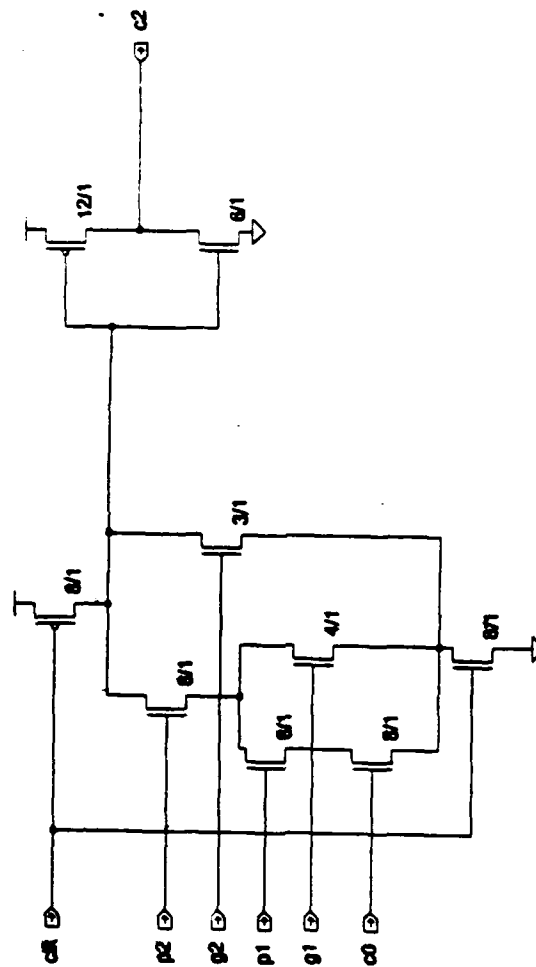
Hannibal 8/15/88 22:06

SPICE:LOOKAHEAD1



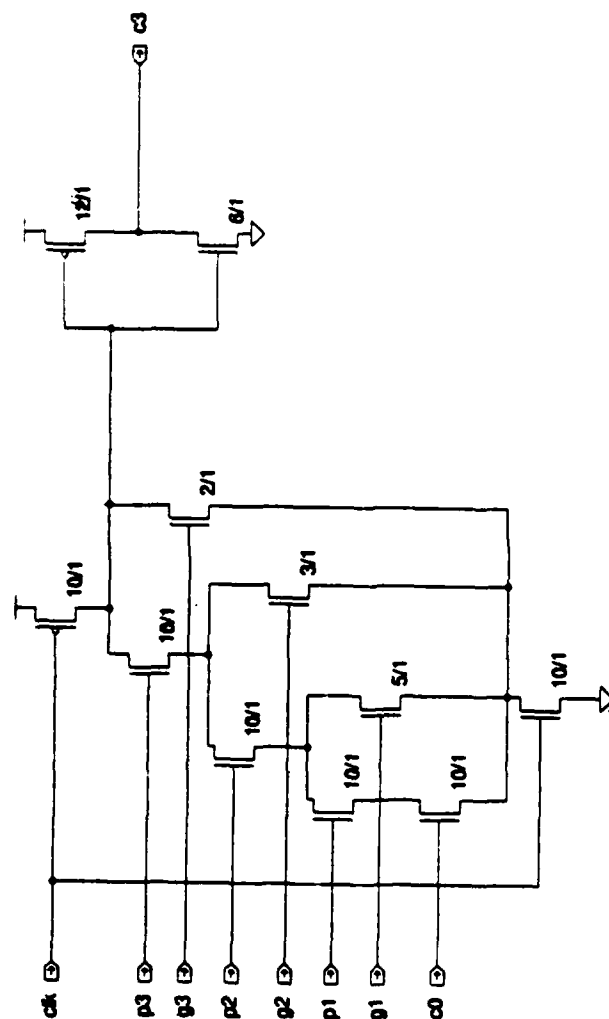
Hannibal | 8/05/88 18:37

SPICE:ADD-LOOK2



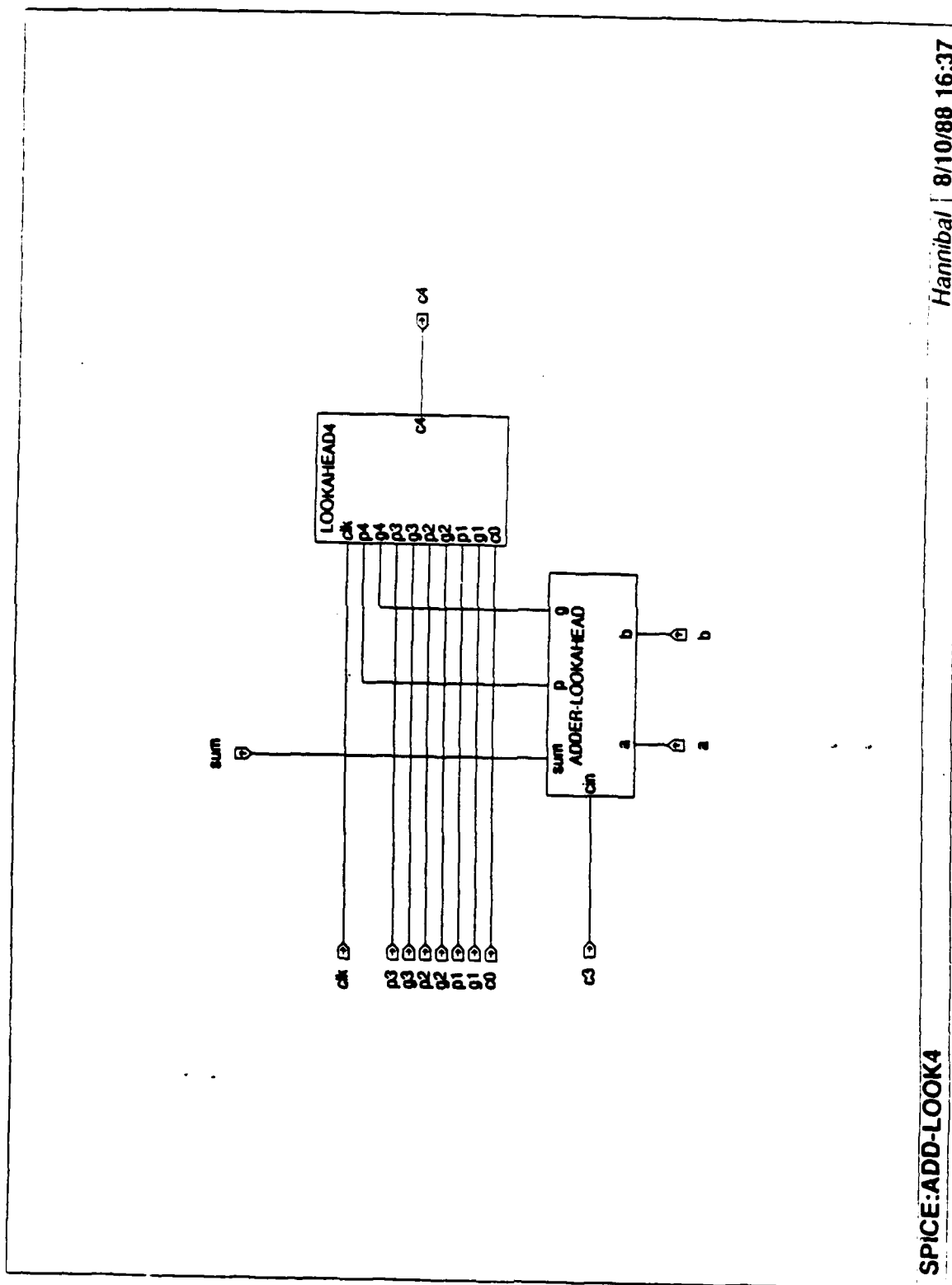
SPICE:LOOKAHEAD2 Hannibal 8/15/88 22:06





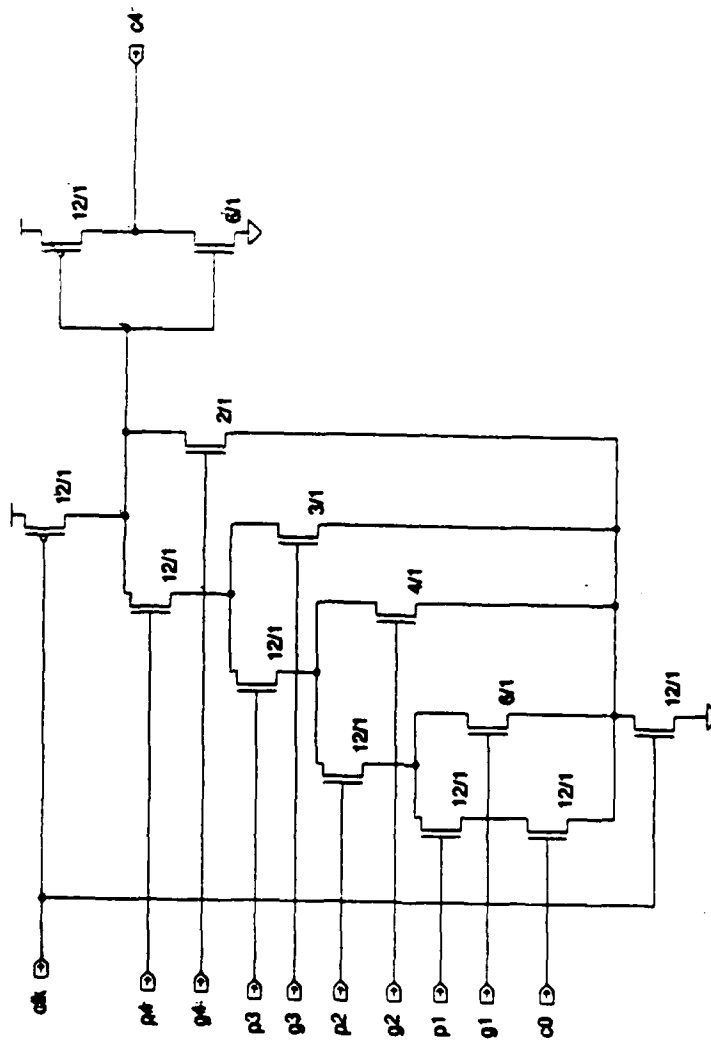
## SPICE: LOOKAHEAD3

**Hannibal** 8/15/88 22:06



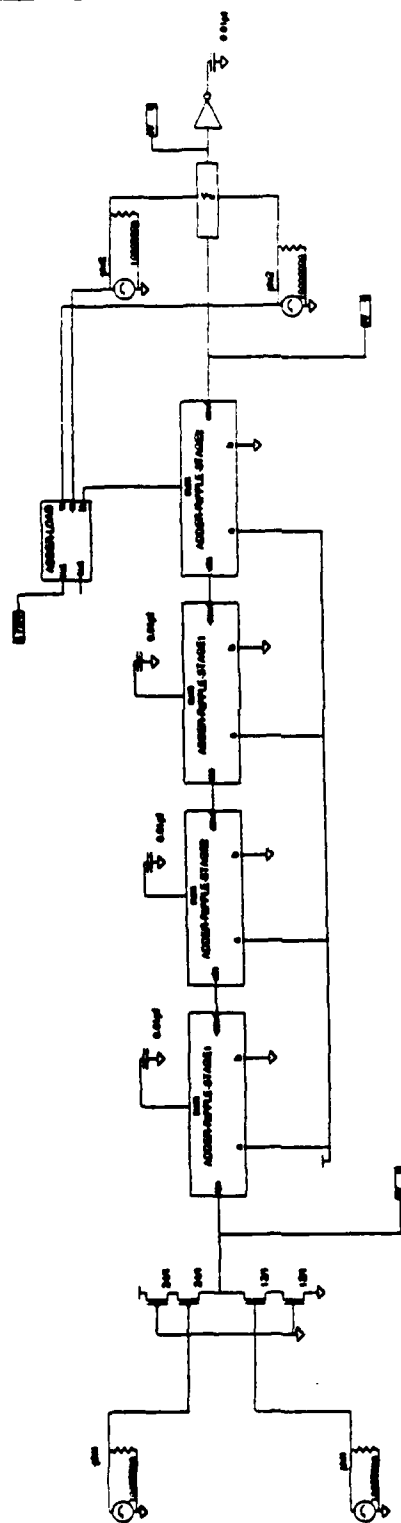
SPICE:ADD-LOOK4

Hannibal 8/10/88 16:37



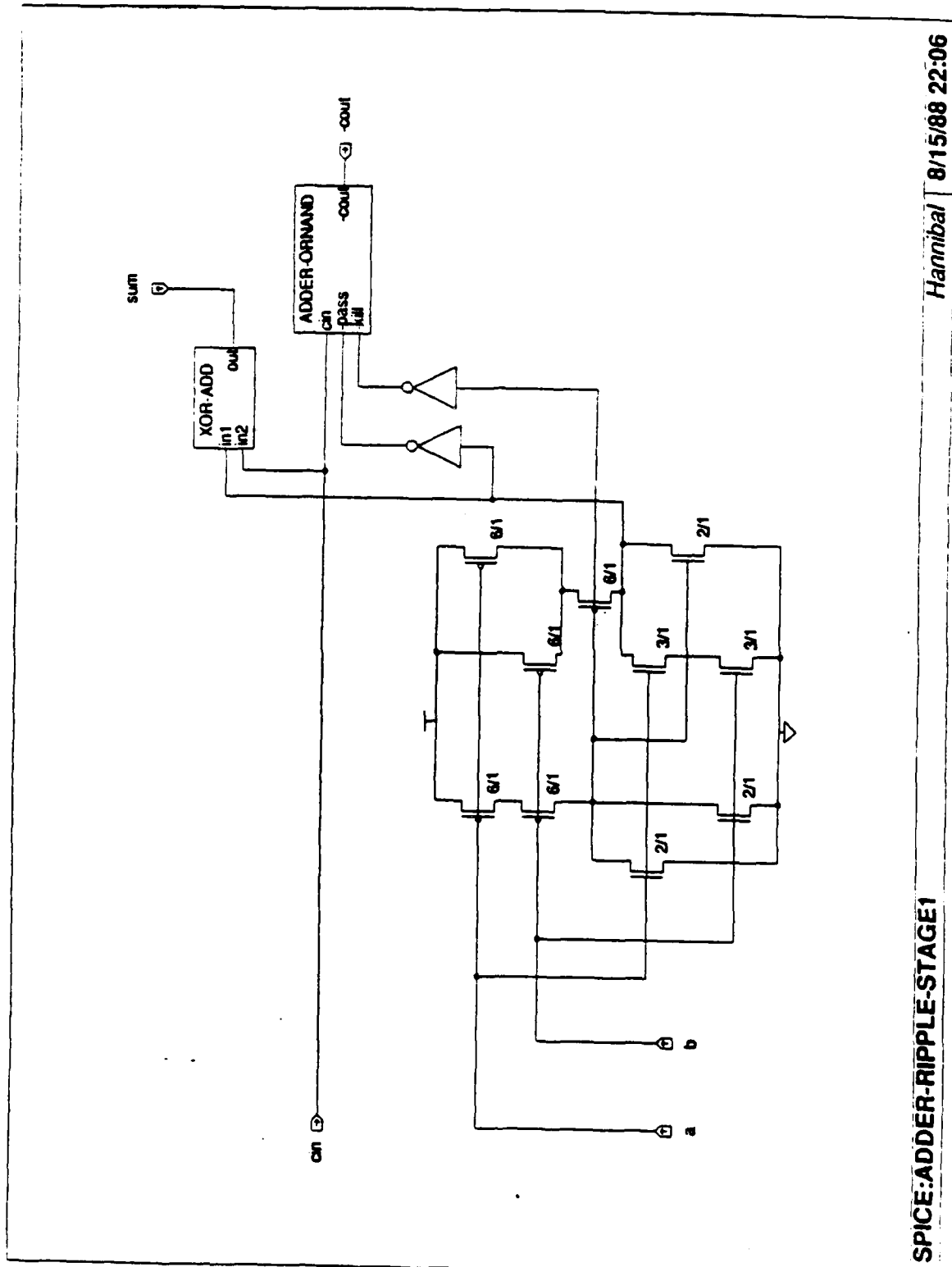
SPICE:LOOKAHEAD4

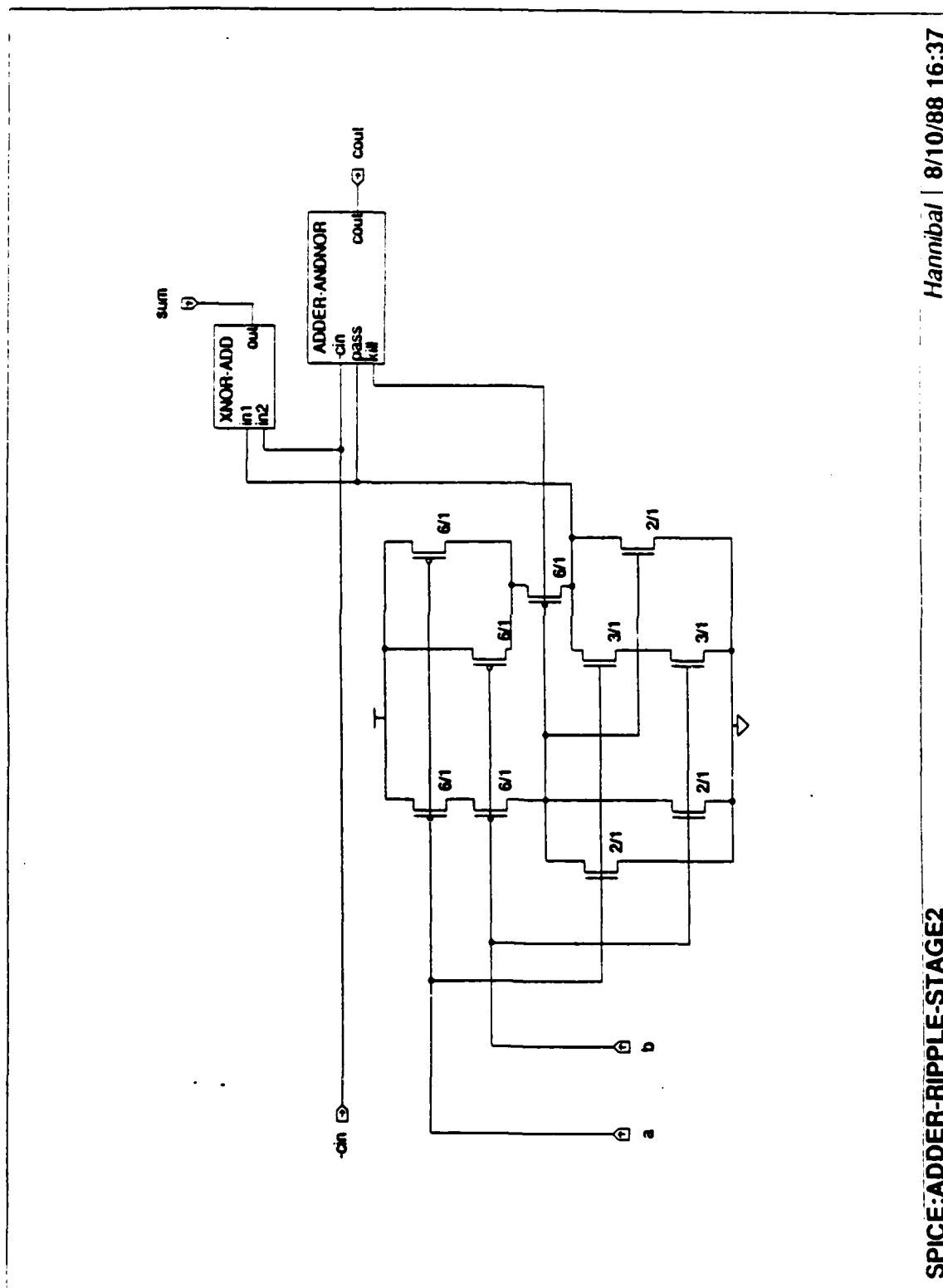
Hannibal | 8/15/88 22:06



**Hannibal | 12/09/88 03:26**

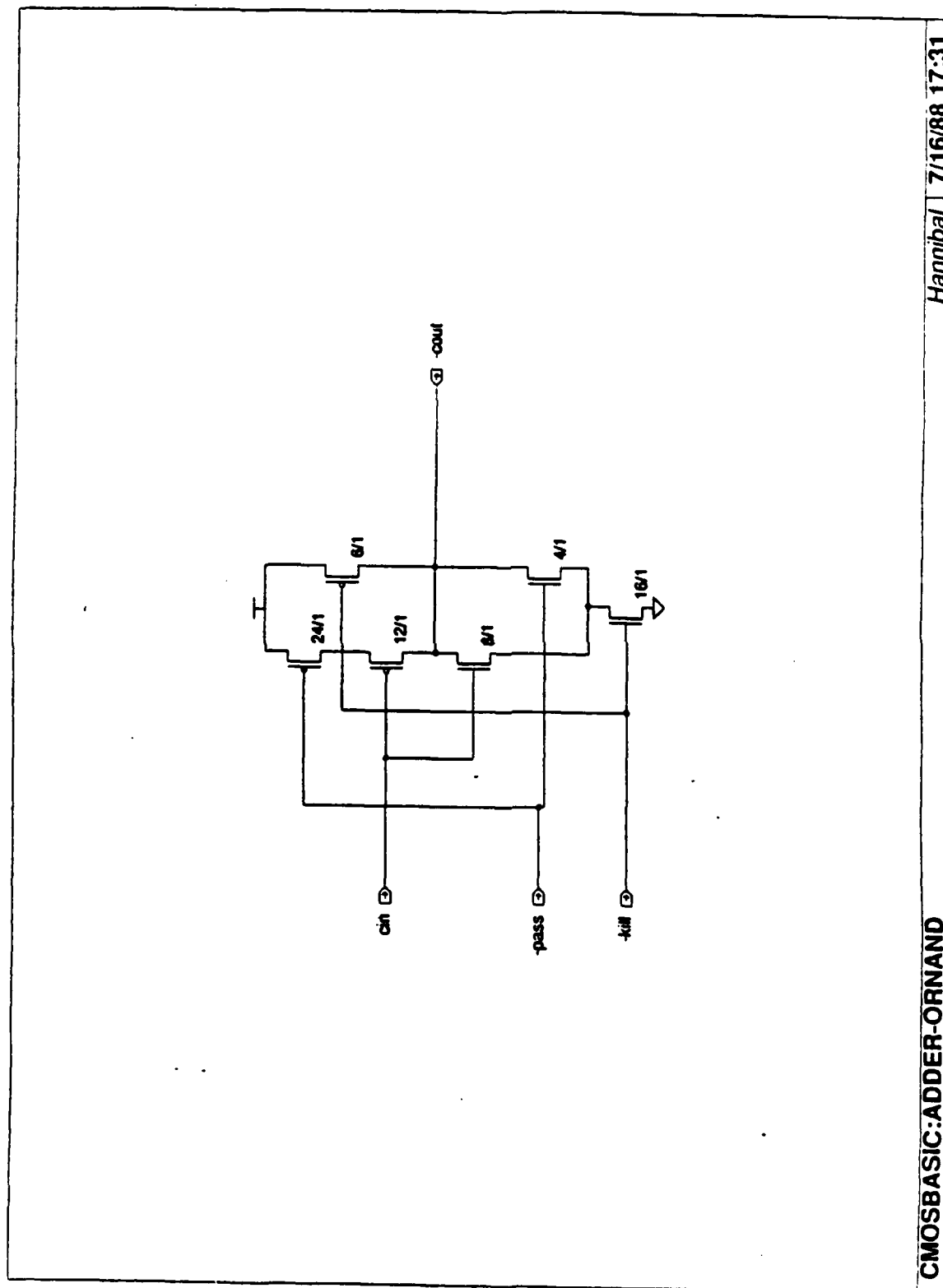
**SPICE:ADLER4**

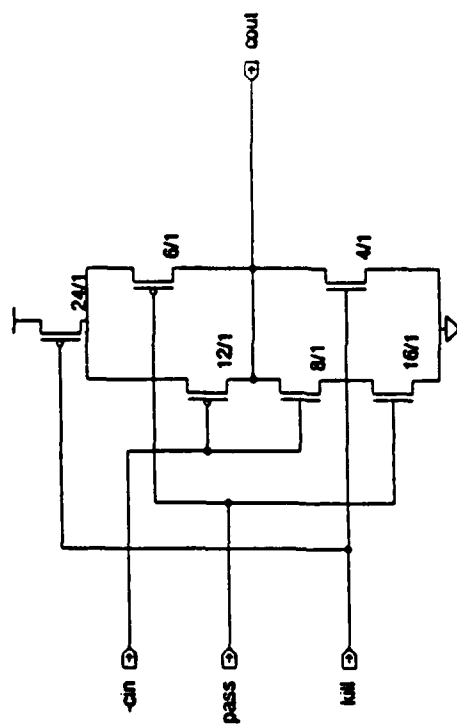




Hannibal | 8/10/88 16:37

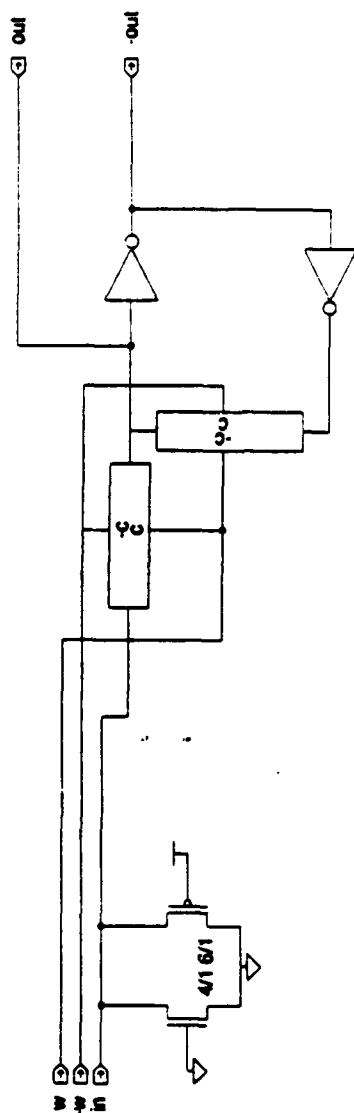
SPICE:ADDER-RIPPLE-STAGE2





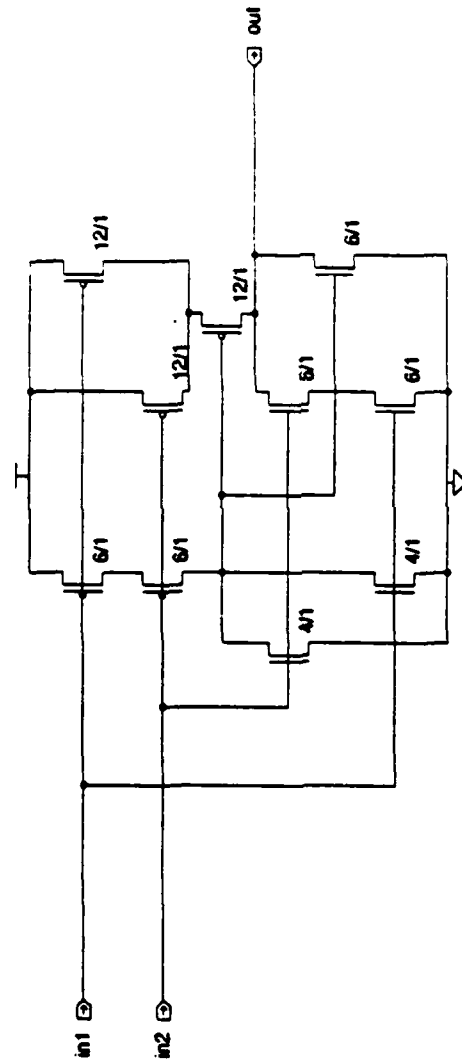
Hannibal 7/16/88 17:31

CMOSBASIC:ADDER-ANDNOR



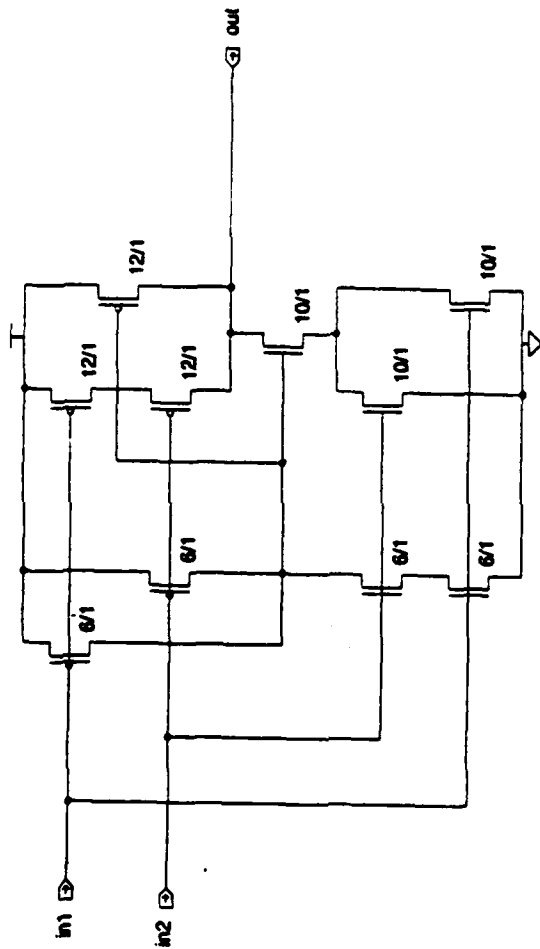
Hannibal | 8/05/88 18:37

SPICE:ADDER-LOAD

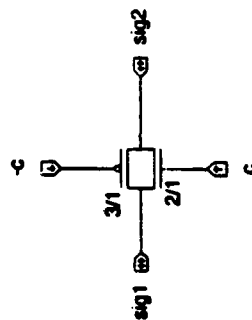


Hannibal 8/07/88 19:13

SPICE:XOR-ADD

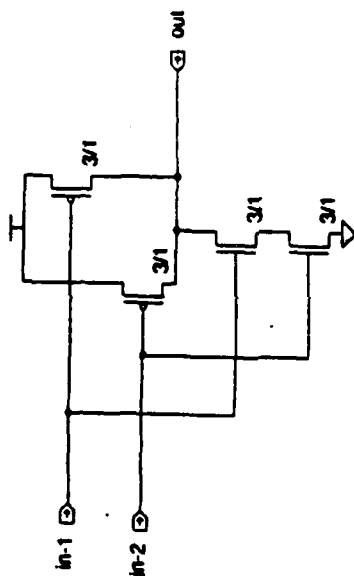


SPICE: XNOR-ADD Hannibal 8/15/88 22:06



Hannibal | 3/27/88 22:54

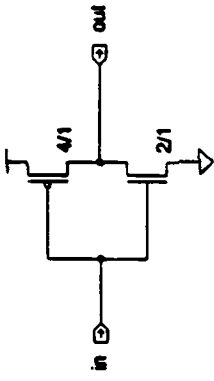
CMOSBASIC:PASST



CMOSBASIC:NAND-2IN

Hannibal

3/27/88 22:53



Hannibal 3/27/88 22:53

CMOSBASIC:INV

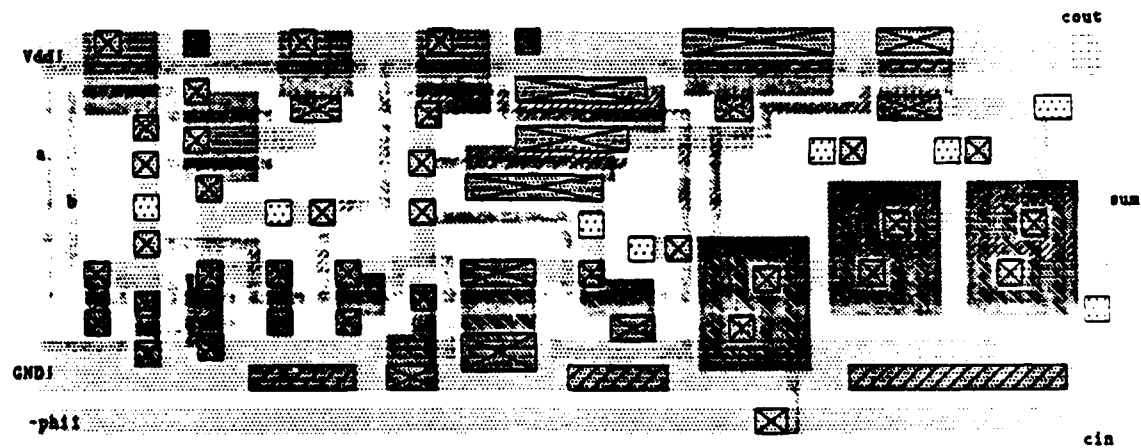
### A.3 4-bit Adder Layout

In this section, plots of the cells used to estimate area are shown. For ADDER1, ADDER2, and ADDER4 a single bit of the adder was layed out, whereas for ADDER3 the complete 4-bit adder is shown, due to the non-uniform nature of the circuit.

## APPENDIX A. 4-BIT ADDER SIMULATION AND LAYOUT

159

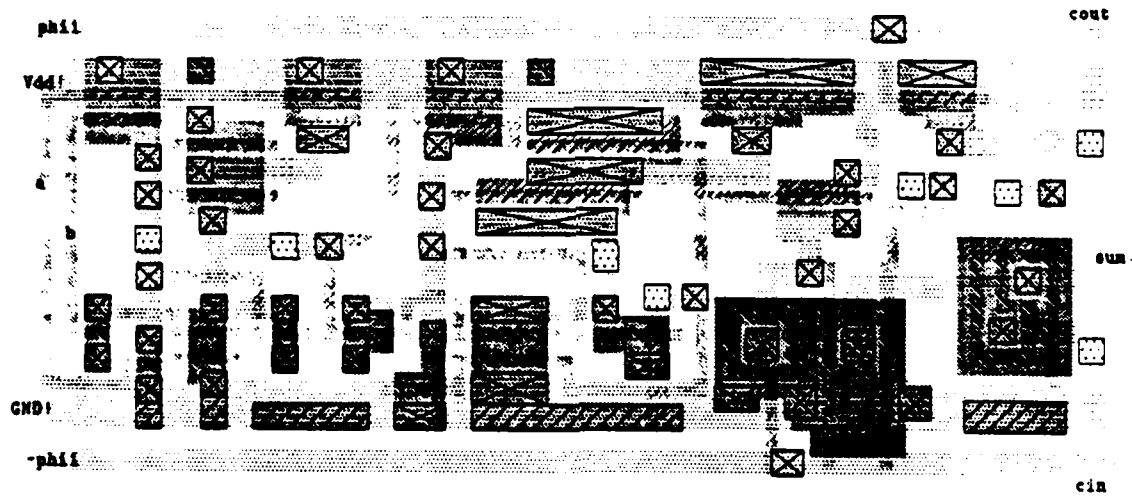
File: adder1  
User: stuart  
Date: Tue Dec 13 00:11:38 1988



# APPENDIX A. 4-BIT ADDER SIMULATION AND LAYOUT

160

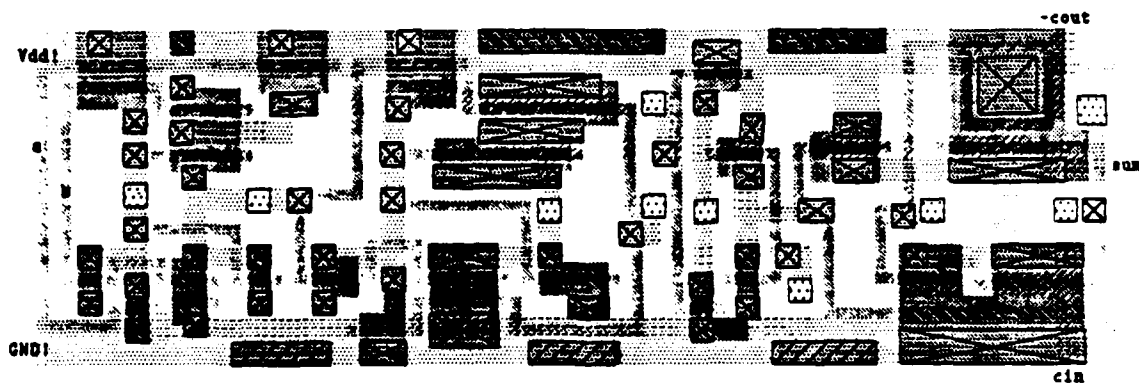
File: adder2  
User: stuart  
Date: Tue Dec 13 09:11:14 1988



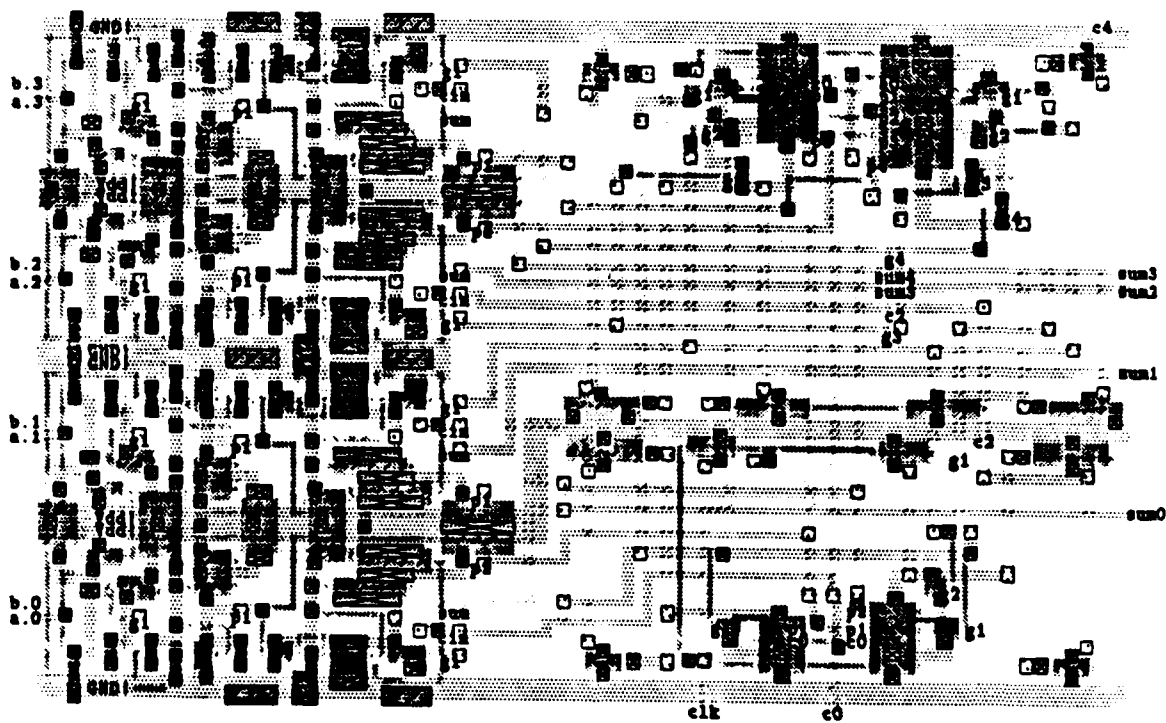
## APPENDIX A. 4-BIT ADDER SIMULATION AND LAYOUT

161

File: adder3  
User: stuart  
Date: Tue Dec 13 09:11:53 1988



File: adder4  
User: stuart  
Date: Tue Dec 20 21:27:08 1988



# Bibliography

- [1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms*. Addison-Wesley Publishing Company, 1983.
- [2] A.D. Booth. A Signed Binary Multiplication Technique. *Q. J. Mech. Appl. Math.*, 4:236-240, 1951.
- [3] J.T. Coonen. An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic. *Computer Magazine*, 13(1):68-79, January 1980.
- [4] Weitek Corporation. WTL 3164/XL-3164, WTL 3364/XL-3364, 1988. Beta Release.
- [5] W.J. Dally. A high performance VLSI quaternary serial multiplier. In *ICCD-87*, pages 649-653, 1987.
- [6] W.J. Dally and Seitz C.L. The Torus Routing Chip. *Journal of Distributed Systems*, 1(3):187-196, 1986.
- [7] W.J. Dally et al. Architecture of a Message-Driven Processor. In *Proceedings of the 14<sup>th</sup> ACM/IEEE Symposium on Computer Architecture*, pages 189-196, June 1987.
- [8] W.J. Dally et al. Concurrent Computer Architecture. In *Proceedings of Symposium on Parallel Computations and Their Impact on Mechanics*, 1987.
- [9] W.J. Dally and Song P.Y. Design of a Self-Timed VLSI Multicomputer Communication Controller. In *ICCD-87*, pages 230-4, October 1987.

- [10] J.R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. The MIT Press, 1986.
- [11] S. Fiske. RAP Expression Compiler Listing. MIT Concurrent VLSI Architecture Memo 18, December 1988.
- [12] S. Fiske. RAP Floating-Point Unit Schematics. MIT Concurrent VLSI Architecture Memo 17, December 1988.
- [13] M.R. Garey and Johnson D.S. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman, 1979.
- [14] Edwards D.B.J. Gosling J.B., Zurwawski J.H.P. A Chip Set for High-Speed Low Cost Floating Point Unit. In *Proceedings of the 5th Symposium on Computer Arithmetic*, pages 50-55. IEEE Computer Society Press, 1981.
- [15] The Institute of Electrical and Electronic Engineers, 345 East 47th Street, New York, NY 10017. *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985 edition, 1985.
- [16] M.J. Irwin and R.M. Owens. Digit-Pipelined Arithmetic as Illustrated By the Paste-Up System: A tutorial. *Computer*, pages 61-73, April 1987.
- [17] H.F. Jordan. HEP Architecture, Programming and Performance. In Kowalik J.S., editor, *Parallel MIMD Computation: HEP Supercomputer and its Applications*, pages 1-40, Cambridge Massachusetts, 1985. MIT Press.
- [18] Hwang K. *Computer Arithmetic: Principles, Architecture, and Design*. John Wiley & Sons, 1979.
- [19] D.J. Kuck. A Survey of Parallel Machine Organization and Programming. *ACM Computing Surveys*, 9(1):29-59, March 1977.
- [20] D.J. Kuck, Y. Muraoka, and S Chen. On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup. *IEEE Transactions on Computers*, C-21(12):1293-1309, December 1972.

- [21] M. Lam. Software Pipelining: An Effective Technique for VLIW Machines. In *Proc. SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318-328, Atlanta, Georgia, June 1988.
- [22] Lucid Inc., 707 Laurel Street, Menlo Park, California 94025. *SUN Common LISP User's Guide*, AAAI edition, 1986.
- [23] R.F. Lyon. Two's Complement Pipeline Multipliers. In *IEEE Transactions on Computers Vol COM-24*, pages 418-425, 1976.
- [24] R.F. Lyon. A Bit-Serial VLSI Architectural Methodology for Signal Processing. In J.P. Gray, editor, *VLSI'81*, pages 131-140. Academic Press, 1981.
- [25] R.F. Lyon. MSSP: A Bit-Serial Multiprocessor for Signal Processing. In *VLSI Signal processing: A Bit-Serial Approach*. Addison-Wesley Publishing Company, 1985.
- [26] W.H. McAllister and J.R. Carlson. Floating-Point Chip Set Speeds Real-Time Computer Operation. *Hewlett-Packard Journal*, pages 17-23, February 1984.
- [27] F.H. McMahon. Lawrence Livermore National Laboratory FORTRAN kernels: MFLOPS, 1984.
- [28] A.V. Oppenheim and R.W. Schaffer. *Digital Signal Processing*. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1975.
- [29] R.M. Owens. Compound Algorithms for Digit On-line Arithmetic. In *5th Symposium on Computer Arithmetic*, pages 64-71, Ann Arbor, MI, May 1981.
- [30] Denyer P. and Renshaw W. *VLSI Signal Processing: A Bit-Serial Approach*. Addison-Wesley Publishing Company, 1985.
- [31] L.W. Rabiner and B. Gold. *Theory and Applications of Digital Signal Processing*. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1975.
- [32] K. Rauch. Math Chips: how they work. *IEEE Spectrum*, pages 25-30, July 1987.

- [33] R. Sedgewick. *Algorithms*. Addison-Wesley Publishing Company, 1984.
- [34] B. Smith. The Architecture of HEP. In Kowalik J.S., editor, *Parallel MIMD Computation: HEP Supercomputer and its Applications*, pages 41-55, Cambridge Massachusetts, 1985. MIT Press.
- [35] K.S. Trivedi and M.D. Ercegovac. On-line Algorithms for Division and Multiplication. *IEEE Transactions on Computers*, C-26(7):681-687, July 1977.
- [36] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design, A Systems Perspective*. Addison-Wesley Publishing Company, 1985.
- [37] P.H. Winston. *Artificial Intelligence*. Addison-Wesley, Massachusetts, 1985.